MASTER EN SCIENCES INFORMATIQUES

Contribution au projet PRISM : porter le mode slave passif du whiteboard sur le web

Bazzi, Mohamed

Award date:
1996

FACULTES UNIVERSITAIRES NOTRE DAME DE LA PAIX, NAMUR
INSTITUT D'INFORMATIQUE

Année académique 1995-1996

CONTRIBUTION AU PROJET PRISM : PORTER LE MODE *SLAVE* PASSIF DU *WHITEBOARD* SUR LE *WEB*

Mohamed BAZZI

Promoteur : Professeur Jean Ramaekers

Mémoire présenté pour l'obtention du grade de Licencié et Maître en Informatique

Résumé

L'objectif du mémoire est de porter le mode ***slave*** **passif** du *Whiteboard* sur le *Web*. Le *Whiteboard* est un outil multimédia de haut niveau conçu dans le cadre du projet PRISM développé à l'**E**cole **N**ationale **S**upérieure de **T**élécommunication de **B**retagne (ENSTB).

PRISM est l'acronyme de **P**late-forme **R**épartie pour l'**I**ntégration de **S**ervices **M**ultimédia. L'objectif de PRISM est de masquer toute complexité survenant lors de la conception d'une application multimédia dans un environnement hétérogène. PRISM se compose de trois blocs : le bloc multimédia où l'on trouve principalement un certain nombre de services de haut niveau (son, image, espace de travail commun...), le bloc de distribution qui autorise le transfert de données multimédia et le bloc de gestion qui gère l'ensemble des services multimédia proposés et demandés.

Le *Whiteboard* est l'un des outils de haut niveau du bloc multimédia. Il offre un espace de travail commun. Il est implémenté en C dans l'environnement UNIX XWindow. Le *Whiteboard* peut être activé selon deux modes : le mode ***master*** et mode ***slave***.

En portant le mode *slave* passif sur le *Web*, chaque utilisateur, via un navigateur doté de l'environnement JAVA, peut télécharger une page HTML contenant un code compilé et l'exécuter localement. Par ailleurs, un prototype a été réalisé pour intégrer le son dans le mode *slave* passif. L'environnement JAVA permet de développer des applications hautement interactives sur le *Web*. En combinant JAVA avec d'autres technologies on peut rendre le *Web* totalement client/serveur.

Abstract

The aim of the thesis is to develop the **passive slave** mode of the Whiteboard application, a high-level software tool conceived in the framework of the PRISM project developed at the **E**cole **S**uperieure de **T**élécommuication de **B**retagne (ENSTB), on the Web using the JAVA environment.

PRISM is the French acronym for «**P**late-forme **R**épartie pour l'**I**ntégration de **S**ervices **M**ultimédia ». The aim of the PRISM is to hide such complexity during the conception of Multimedia application in a heterogeneous environment. PRISM is composed of three blocks : the multimedia block where we find high level tools (sound, picture, common work space...), the distribution block which allows the transfer of multimedia data and the management block which manages all requested and available multimedia services.

The Whiteboard is one of the high level tools of the multimedia block. It offers a common work space. It is implemented in C in the UNIX XWindow environment. The Whiteboard supports two modes : the master mode and the slave mode.

Our work was to make the **passive slave** mode of the Whiteboard available over the *Web* so that any user via a browser doted with the JAVA environment can download a HTML page containing a compiled code and execute it locally. A prototype has been realized to integrate the audio in the **passive slave** mode of the Whiteboard. JAVA environment permits the development of applications highly interactive over the Web. Using JAVA with other technologies, the Web can be made totally client/server.

AVANT-PROPOS

Je voudrais remercier l'ensemble des personnes qui m'ont aidé
à réaliser ce mémoire,
Monsieur le professeur Jean Ramaekers qui m'a proposé le sujet
et a suivi sa réalisation avec beaucoup d'attention,
Monsieur le professeur Pierre Rolin qui m'a accueilli au département
réseau et multimédia de l'ENSTB de Rennes,
Monsieur LaurentToutain pour son aide précieuse durant mon stage
à Rennes,
Oliver Huber qui m'a aidé à comprendre le Whiteboard,
Monsieur Radu Cotet pour l'intérêt qu'il a porté à ce travail,
Pascal Goossens pour ses remarques judicieuses,
Mes parents pour leur soutien moral et financier durant mes études.

Table des matières

Chapitre 1 : INTRODUCTION

Le mémoire s'inscrit dans le cadre du projet **PRISM** développé à l'**E**cole **N**ationale **S**upérieure de **T**élécommunications de **B**retagne (ENSTB). PRISM (présenté au chapitre 2) est l'acronyme de **P**late-forme **R**épartie pour l'**I**ntégration de **S**ervices **M**ultimédia. Le but de PRISM est de permettre le développement d'applications multimédia dans un environnement hétérogène.

Notre contribution était de porter une partie de l'application ***Whiteboard*** (un outil de haut niveau de PRISM) sur le ***Web***. Le but est de permettre à n'importe quel utilisateur par le biais d'un navigateur *Web* de télécharger une page HTML contenant cette partie de l'application et de l'exécuter localement. L'application *Whiteboard* (présenté au chapitre 6) permet à un conférencier ou à un professeur de montrer des slides (images préenregistrées) et d'accomplir un certain nombre d'actions sur ces slides. L'application *Whiteboard* peut être activée selon deux modes : le mode ***master*** et le mode ***slave***. Le mode *master* est typiquement le cas du conférencier qui exécute des actions et prononce un discours. Le mode *slave* est celui de l'auditeur qui visualise les actions du conférencier et entend son discours.

C'est le mode ***slave*** **passif** de l'application que nous avons porté sur le *Web*. Dans le mode ***slave*** **passif**, l'auditeur ne peut pas intervenir. Par ailleurs nous avons réalisé un prototype pour intégrer le son dans le mode ***slave*** **passif** du *Whiteboard*. Les deux applications réalisées sont présentées au chapitre 7.

L'outil de développement utilisé est le langage **JAVA** et son concept de **Machine Virtuelle**. Le langage JAVA (présenté au chapitre 5) s'inscrit dans le contexte de cette mutation qui existe actuellement à savoir d'une part la bataille pour l'**Infrastructure Internet**, dans ce cadre JAVA représente un enjeu considérable, et d'autre part l'évolution que l'on constate des applications client/serveur classiques (présenté au chapitre 3) vers des architectures client/serveur plus ouvertes utilisant la technologie des **objets distribués** (présenté au chapitre 4) et profitant des différentes opportunités offertes par le *Web* . Une partie du mémoire est consacrée à l'étude des technologies qui existent actuellement, chacune possédant ses propres caractéristiques mais toutes s'inscrivent dans le cadre du développement des réseaux locaux (LAN) vers des réseaux à grande distance (WAN).

Chapitre 2 : LA PLATE-FORME PRISM

2.1 Introduction

A l'instar du modèle OSI, **PRISM**[1] propose une architecture, c'est-à-dire une organisation, pour les services multimédia. Cette architecture générale permet d'inclure un grand nombre de services (diffusion de données multimédia, groupware, hypermédia,...). PRISM permet aussi de tester des idées originales en ce qui concerne les algorithmes de routage, l'administration des applications multimédia distribuées et les protocoles de transport.

L'approche PRISM se veut pragmatique. A partir des moyens actuels dont on dispose sur les stations de travail et sur les PCs et l'infrastructure existante (réseaux, capacité des systèmes de fichiers), PRISM permet d'écrire rapidement des applications. Dans l'état actuel des choses les performances sont limitées (débit faible, interactivité réduite) du fait des techniques (réseaux, systèmes répartis utilisés, ...). Des services de téléséminaires (cours ou séminaires diffusés par le réseau ou archivés sur disque) sont adaptés à ce type de contraintes. Mais l'approche prise par PRISM se veut évolutive, lorsque les techniques offriront des services plus performants, PRISM sera prêt à les accueillir.

PRISM est organisé selon 3 blocs (figure 2.1) à savoir le bloc de gestion, le bloc multimédia et le bloc de distribution :

2.2 Le bloc de gestion

L'architecture de ce bloc n'est pas finalisée. Elle regroupe des services locaux ou répartis servant à la gestion de la plate-forme. Ces services s'appuient sur des normes ou des standards de fait (X500, SNMP, IP, ...).
Mais avant d'étudier les services offerts par le bloc de gestion on définit les concepts suivants : espace, canal et flot.

. Un **canal** est une adresse spécifique dans un **espace** qui contient des informations multimédia.

Un canal peut être :

. Une adresse Internet de classe A, B, C ou D (PRISM_INTERNET),
. Un nom de fichier (PRISM_FILE),
. Un numéro de téléphone (PRISM_ISDN).

[1] [Hub94] p : 3-9, [Van95] p : 7-15

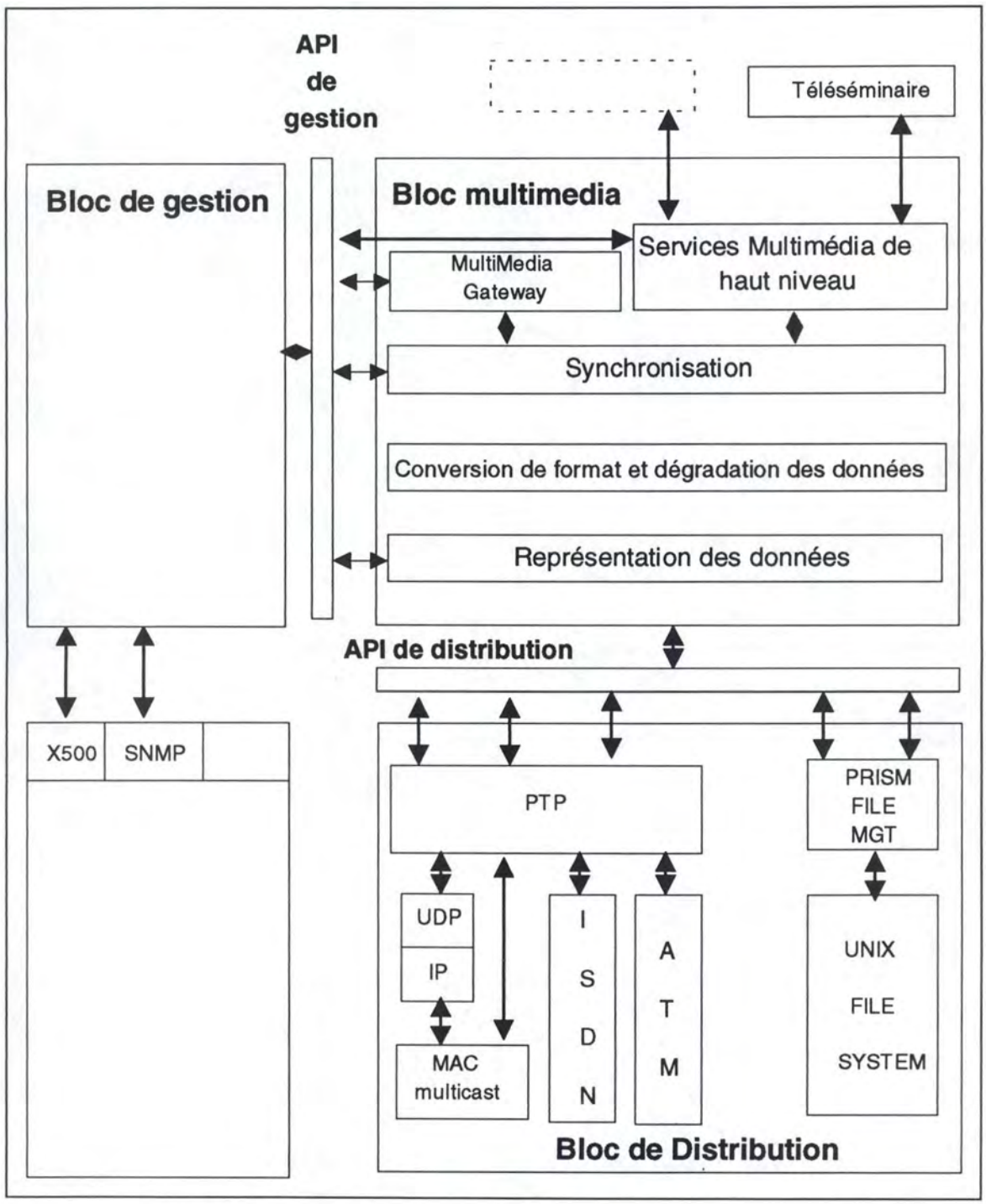


Figure 2.1 : Architecture de la plate-forme PRISM

. L'**espace** est défini en un ensemble de canaux.
. Un **flot** est une partie de canal qui contient des informations monomédia, celles-ci peuvent être multiplexées. Un flot est en mode simplex (unidirectionnel). Il est identifié par un numéro, un canal, et pour des raisons de sécurité par une adresse d'origine. Actuellement, ces numéros sont assignés de façon statique mais dans des versions ultérieures, de façon à gérer l'évolution des standards et l'hétérogénéité, cette assignation sera dynamique.

Voici à présent les services que l'on souhaite trouver :

. **Attribution des canaux** ou des **flots** pour un domaine **multicast** donnée. Le multicast désigne le nombre de stations potentiellement atteintes par un seul paquet. Dans des applications telles que les téléséminaires, le multicast est très important puisque le conférencier délivre un seul paquet qui suivant la topologie du réseau est dupliqué vers plusieurs auditeurs.
Ce service permet de créer de nouveaux canaux ou flots. Il a une connaissance des adresses utilisées dans le domaine de multicast soit par les applications PRISM, soit par les autres applications.
. Un service assurant la **gestion du routage** ou de **connexion** à des services existants. La figure 2.2 montre un exemple de routage. Le site 4 veut se connecter à un service diffusé par le site 1. Les sites 2 et 3 rediffusent les données du site 1. Avec des requêtes X500, le bloc obtient la liste des sites diffusant les données multimédia ainsi que la qualité de service associée à chaque domaine (continuité du service, localisation géographique, topologie du réseau, délai de propagation, coût de transmission, qualité des données,...). A partir de ces critères, le service de connexion se connecte au site choisi.

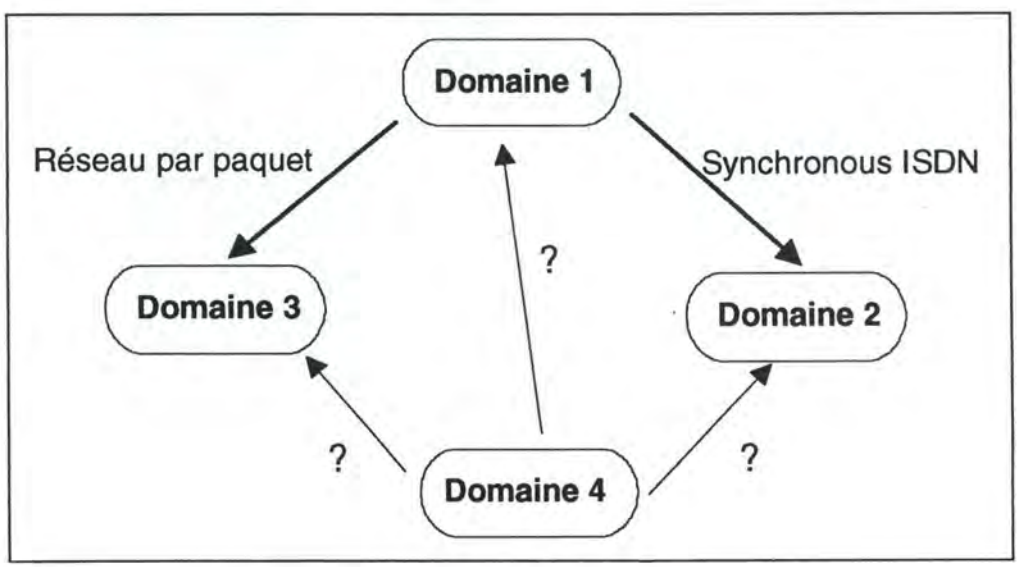


Figure 2.2 : Gestion de routage

. **Service de supervision**. La bande passante offerte par le réseau doit être partagée entre toutes les applications (PRISM ou non). Le service de supervision influe sur la représentation des données et la taille des paquets pour que la qualité de service, en particulier la régularité des flux, soit respectée. Pour cela il interroge des agents SNMP pour connaître la charge du réseau, le nombre de collisions, le temps de rotation du jeton, le nombre de paquets rejetés par les routeurs,...

2.3 Le bloc multimédia

Au sommet du bloc multimédia, on trouve **une API application** qui permet d'appeler et de paramétrer les différents services offerts par PRISM. Ces services, au début primaires, seront améliorés au fil du temps. Par exemple :

. **Services pour le son** : un service de production et un service de consommation des données sonores sont définis.

. **Services pour l'espace commun** : un service qui peut à la fois servir d'affichage passif ou de pilotage avec comme fonctionnalités :

- possibilité d'afficher une image dans un format quelconque (JPEG, GIF, ...)
- affichage d'un pointeur indiquant les points importants
- zone sensible captant la position du pointeur du conférencier
- des outils de dessin et d'écriture
- des outils d'interrogation des auditeurs, permettant de donner le contrôle des flots par un auditeur
- de gestion complètement décentralisée pour le travail coopératif.

Au même niveau se trouve le **M**ulti**M**edia **G**ateway (MMG). Les MMG forment un service réparti chargé de rendre accessible à un utilisateur (i.e. aux services de haut niveau) une information qui lui est inaccessible : soit que cette information n'est pas transmise sur le réseau, soit que le format de la représentation des données est incompatible avec le matériel disponible sur la station.

Par exemple une MMG peut permettre à des services fonctionnant sous différentes technologies d'interopérer. On peut imaginer des MMG effectuant les transformations suivantes :

. **téléamphi vers téléséminaire** en traduisant les données provenant du réseau RNIS vers le réseau Internet. Les fonctionnalités étant à peu près identiques dans les deux services l'adaptation est assez simple.

. **téléséminaire vers ivs**[1]. Un téléséminaire est basé sur la notion d'espace commun dans lequel sont affichés les transparents de la conférence. Dans ivs les transparents sont filmés par une caméra et transmis en séquence animée en utilisant le protocole H.261. La MMG fera la transformation du fond (i.e. transparent) et du mouvement (déplacement du pointeur) en séquence H.261.

. **audio vers téléphone** en connectant une station ayant PRISM à un PABX.

La MMG peut aussi utiliser les services de la couche de conversion pour dégrader la qualité des données multimédia et les adapter aux caractéristiques des voies de communication et ainsi garantir les contraintes temporelles.

2.4 Le bloc de distribution

Traiter des données multimédia, oblige les applications à les consommer au rythme où elles ont été produites. Ceci se traduit par un changement dans la philosophie avec laquelle les protocoles des réseaux par paquet ont été pensés. Ceci est dû à la perception humaine de l'information qui est surtout sensible aux erreurs temporelles (i.e. retard de l'information). S'il est possible de changer en partie les protocoles qui se trouvent sur un réseau local, il n'en est pas de même pour les MAN et les WAN. Malgré la perte d'un paquet dans la transmission du son, un message restera compréhensible mais si le protocole essaie de retransmettre le paquet, il introduit une désynchronisation et le son devient non interprétable. Pour compenser la gigue introduite par ceux-ci, une technique simple consiste en la mémorisation d'une certaine quantité d'information chez les récepteurs pour qu'ils aient toujours des informations à fournir à l'utilisateur même en cas de retard dans la transmission qui en comporte déjà beaucoup (mémorisation chez l'émetteur pour former un paquet, mémorisation dans les noeuds intermédiaires, retransmission des capacités d'interactivité des applications). En fonction du coût financier acceptable, le retard sera un des facteurs importants dans le choix d'un réseau synchrone (qui n'introduit que très peu de retard) ou d'un réseau par paquet.

Il en résulte que PRISM doit être capable de s'adapter à différentes technologies pour le transfert de l'information multimédia. Le point commun à toutes ces méthodes de distribution de l'information multimédia concerne des flux mono-directionnels et monomédia.

[1]Ivs est l'acronyme de "INRIA videoconferencing system", developpé à l'INRIA, Nice, qui permet des transmissions video avec le code H.261 à travers le protocole multicast dans Internet

Donc de façon à ne pas limiter la plate-forme à certains environnements, plusieurs espaces ont été définis, un espace étant un moyen de transférer de l'information d'un point à un autre.

Dans le projet, les espaces suivants existent :

. PRISM_INTERNET : le protocole utilisé est UDP/IP.
. PRISM/FILE : les données sont stockées en utilisant le système de fichier, ce qui permet, par exemple de réutiliser une conférence préalablement enregistrée. C'est cet espace que l'on a utilisé pour porter le mode *slave* passif du *Whiteboard* sur le *Web*.
. PRISM_PRISM : un protocole adapté au multimédia est utilisé, il est implémenté juste au-dessus de la couche MAC.
. PRISM_IVS : le protocole de transmission IVS est implémenté au-dessus de UDP/IP.
. PRISM_ISDN : le protocole utilise un lien ISDN.
. PRISM_ATM : des exigences de grands débits seront pris en compte.

2.5 Les APIs de la plate-forme PRISM

Les API permettent une indépendance vis-à-vis du système et entre les différents blocs de PRISM. Il s'agit pour l'essentiel de bibliothèques d'interfaces garantissant une pérennité des blocs lors des évolutions et des développements futurs. L'**API application** pour l'appel des services de haut niveau n'est pas encore défini précisément. L'**API de distribution** est orientée autour de la notion de flot. Il existe des commandes pour créer ou écouter des flots sur des canaux.

L'adresse des canaux ou les numéros de flot sont donnés par le bloc de gestion. Une commande PrismPilotChannel a été définie pour contrôler certains flots. Pour les données stockées sur disque, cette fonction permet de choisir la séquence à diffuser. Ceci peut être utilisé pour des applications hypermédia.

L'**API de gestion** regroupe les API pour les différents services de gestion. Pour la gestion des canaux et des flots, des fonctions d'enregistrement et de demande de flots en fonction de critères particuliers (normes, QoS, ...) sont en cours de définition. Par exemple lorsqu'une application demande un flot ou un canal, cela peut conduire à l'activation d'une MMG sur le domaine pour la traduction des données dans le bon format ou pour trouver les données sur un autre site. Ces services de gestion sont transparents aux services du bloc multimédia.

2.6 Le protocole de transmission PRISM (PTP)

Le protocole de transmission PRISM (**PTP**) est une simple manière de transmettre les données. Ce protocole est adapté à un LAN. Pour d'autres réseaux, comme ISDN, d'autres protocoles doivent être définies. On suppose que :

. Le réseau offre des facilités multicast (les transmissions sont de type 1-N) et l'utilisateur ignore quelles sont les stations qui reçoivent les données.
. Le réseau est fiable et l'émetteur recevra des acquittements négatifs uniquement si un paquet est perdu.
. L'accès au réseau est déterministe.
. Le protocole réseau peut gérer les priorités.

Le protocole est implémenté au-dessus de la couche MAC et il est utilisé pour transférer des informations ayant des contraintes temporelles.

PTP a été conçu dans le but d'exploiter la propriété de tolérance aux pertes que présentent les données multimédia. On distingue au sein du flux d'information un trafic tolérant les pertes et un trafic sensible. Un mécanisme de fiabilisation peut être appliqué au second.

Par ailleurs, le protocole permet un découpage des débits d'arrivée et de consommation des données. Ainsi, des machines différentes peuvent recevoir des versions plus ou moins dégradées des informations au sein d'un même domaine PRISM sans que la qualité de réception ne soit affectée.

PTP est structuré en trois niveaux qui sont chargés de diffuser l'information, de fiabiliser la diffusion et finalement de stocker et délivrer les données aux applications réceptrices.

- Diffusion

Le choix de pratiquer la diffusion (multicast) est dicté par le type des applications multimédia supportées qui impliquent plus de deux utilisateurs. Par ailleurs, lorsque cette diffusion est réalisée au niveau un ou deux, l'économie de ressources réseau n'est pas négligeable.

- Fiabilité

Les données multimédia sont par nature résistantes aux pertes, suite à la redondance qui les caractérise mais également suite à la nature du récepteur final (l'être humain). La résistance aux pertes est toutefois maximale pour les données brutes (sans compression).

Plusieurs techniques sont utilisables pour fiabiliser les communications : acquittement et retransmission ou techniques préventives (introduction d'une redondance contrôlée des données). Le choix posé ici est d'utiliser des acquittements négatifs et de retransmettre les paquets perdus. Le contrôle des pertes est réalisé par numérotation et il est effectué de façon particulière par PTP. Il est en effet possible d'inclure dans chaque paquet du trafic non fiable la numérotation des paquets devant être absolument reçus. Lorsqu'une perte est détectée, chaque récepteur du flot émet un acquittement négatif.

L'entité émettrice traite le premier acquittement reçu en envoyant de nouveau les paquets puis ignore les autres requêtes relatives aux mêmes pertes pendant un certain délai.

- Gestion de la mémoire

Les protocoles traditionnels sont destinés à des communications point-à-point sans contrainte temporelle, une seule entité réceptrice peut consommer les données stockées par le protocole et elle les consomme à son propre rythme. La gestion en mémoire de ces données est organisée selon le principe d'une file d'attente avec un contrôle de débordement impliquant, lorsque la file est saturée, la perte des données entrantes. Cette politique est inadaptée aux données multimédia car :

. Plusieurs entités de niveau application peuvent recevoir les mêmes données

. Un flux multimédia présente de fortes contraintes temporelles et la politique traditionnelle favorise les données obsolètes.

Il faut donc un mécanisme de stockage des données entrantes qui soit susceptible de délivrer les informations à un nombre arbitraire d'applications.

Les contraintes temporelles sont respectées en découplant le débit d'arrivée des données du débit de consommation : un flux multimédia peut être considéré comme une séquence d'unités d'information (**UI**), une **UI** étant un ensemble de données partageant une sémantique commune du point de vue de leur traitement par une application (exemple : une suite d'images MPEG).

Pour la transmission, les UI seront morcelées en plusieurs paquets. Chaque paquet contient ainsi le numéro de l'UI et la valeur du décalage dans celle-ci.

2.7 L'intégration des applications dans la plate-forme PRISM

Actuellement l'application principale, est le **téléséminaire**. Cette application est utilisée pour distribuer, stocker et rejouer les cours et les conférences. Le conférencier est dans un studio, un show-room ou devant sa propre station de travail. Les auditeurs sont aussi devant leur propre station de travail ou dans une salle de classe. Le conférencier utilise des documents digitalisés (slides). Les auditeurs obtiennent une copie des slides avant le début de la conférence (ou du cours). La voix et les actions du conférencier (pointeur de souris, dessin, ...) sont envoyées en "temps réel" à l'audience. Cette application a été choisie car elle n'engendre pas de volume de données très élevé sur le réseau et la synchronisation entre les différents objets multimédia n'est pas significativement importante.

Notre participation au projet a été de porter le mode ***slave*** **passif** de cette application (c'est-à-dire le cas de l'auditeur qui visualise les actions du conférencier et entend son discours mais sans possibilité d'intervention) sur le *Web*. Donc par rapport à la plate-forme PRISM on a travaillé sur le **PRISM FILE MGT** (pour gérer les cours enregistrés). La partie distribution est prise en compte par les ***Applets*** **JAVA**. Les fonctionnalités du bloc de gestion ne sont pas utilisées.

2.8 Conclusion

PRISM via les 3 blocs constituants son architecture permet la gestion de données multimédia (son, images et vidéo) dans un environnement hétérogène caractérisé par une variété de réseaux, de logiciels. Ces 3 blocs sont : le **bloc multimédia** qui offre des services de haut niveau (son, image animée, espace de travail,...), le **bloc de distribution** qui s'occupe du transfert des données multimédia et le **bloc de gestion** qui gère l'ensemble des services multimédia proposés et demandés.

PRISM cache donc les complexités qui peuvent survenir lors du développement des applications multimédia tout en maintenant un service de qualité.

Chapitre 3 : L'architecture Client/Serveur

3.1 Introduction

Avant de présenter l'outil utilisé (c'est à dire l'environnement **JAVA**) pour développer le mode ***slave*** **passif** de l'application ***Whiteboard***, il serait bon d'étudier dans quel contexte se situe cet outil. En effet avec le développement du ***Web*** et la diminution du coût des réseaux WAN et MAN, on constate un désir croissant de porter les applications développées dans un environnement fermé et souvent propriétaire dans la plupart des **LAN** vers un environnement plus ouvert qui dépasse les frontières des entreprises utilisant de ce fait soit le réseau mondial **Internet** et donc le *Web* soit des réseaux à large bande tel ATM.

Deux options peuvent être envisagées : utiliser le *Web* et donc le protocole **HTTP** pour faire du client/serveur et c'est **JAVA** ou faire du client/serveur objet en utilisant la technologie des objets distribués et c'est : **CORBA(ORB)** et **OLE(COM/DCOM)**.

Nous allons présenté JAVA dans le chapitre 5, CORBA (ORB) et OLE (COM/DCOM) dans le chapitre 4 .

Dans ce chapitre on définit le modèle **Client/Serveur** et les paradigmes existants.

3.2 Définition

Modèle Client/Serveur[1] **:** Le modèle client/serveur est un concept décrivant la communication entre des consommateurs de service (clients) et des fournisseurs de service (serveurs). Dans la figure 3.1 on présente un simple modèle client/serveur. Le modèle de base est celui dans lequel un client initie une interaction avec un serveur en envoyant un message ou en invoquant une opération; on parle alors de **requête** émise par le client (étape 1). Le client attend la **réponse** du serveur (étape 2), la figure 3.1 montre le cas d'un modèle synchrone. Le serveur de son côté reçoit la requête (étape 3), l'exécute (étape 4) et envoie la réponse au client (étape 5), aprés la terminaison de la période d'attente, le client reçoit la réponse (étape 6).

Il y a d'autres modèles asynchrones qui permettent l'échange de messages entre clients et serveurs par l'intermédiaire de files d'attente.

[1] [Uma93] p : 247-249

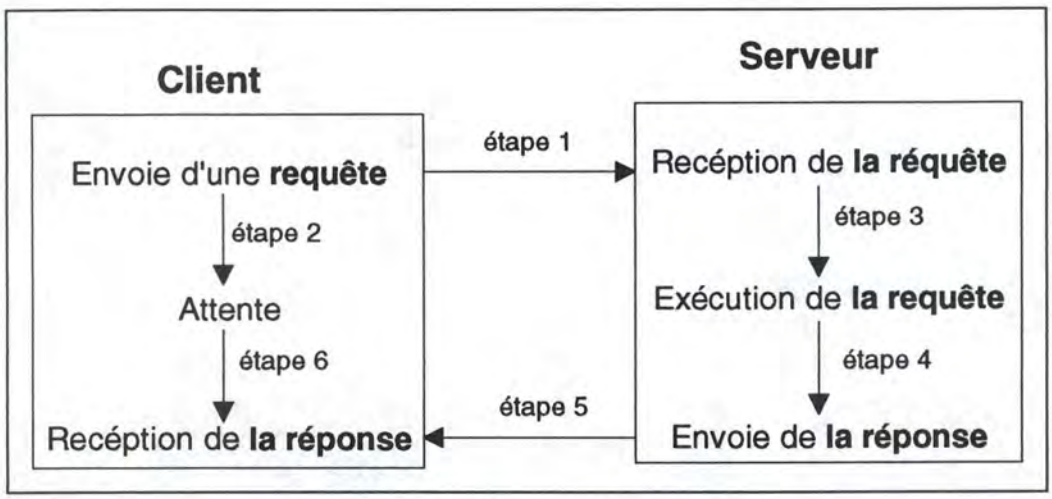


Figure 3.1 : Modèle client/serveur de base

Les clients et les serveurs :

. Sont des processus parallèles indépendants.
. Peuvent tourner sur la même machine ou sur des machines séparées.
. Peuvent tourner sur des machines spécifiques(le terme de serveur est couramment utilisé dans les LANs pour désigner une machine dédier à fournir des services LANs).
. Peuvent avoir besoin de serveurs synchronisés pour la duplication des données et la gestion des incidents.

Mais pourquoi utilise-t-on des systèmes Client/Serveur ?

L'avantage principal du client/serveur est que des processus distants peuvent coopérer ensemble pour réaliser des tâches. Par ailleurs le coût du MIPS des PCs et des stations de travail dans un LAN est faible par rapport à celui des mainframes dans un système centralisé.

Mais ceci peut être compliqué par des problèmes d'ordonnancement, des protocoles réseau, gestion des erreurs et les contraintes de performance. Il y a donc plusieurs facteurs à prendre en considération avant de décider une migration vers un système client/serveur.

Parmi ces facteurs on trouve :

1. La difficulté de déterminer le trafic du réseau.
2. Le coût des logiciels puisqu'on doit installer les logiciels d'interface client sur chaque site client.
3. La nécessité d'avoir des bandes passantes larges pour échanger les messages temps-réel.
4. La gestion efficace du réseau pour minimiser les défaillances du réseau et leurs coûts.

Mais avant de construire un système Client/Serveur, il faut se poser plusieurs questions à savoir :

1. Le degré de la complexité du modèle à construire? La complexité du modèle C/S dépend de plusieurs facteurs notamment si le serveur prend en charge les mises-à-jour des données ou s'il exécute d'autres fonctions de synchronisation.
2. Quelles sont les fonctions exécutées par les clients versus les serveurs ?
3. Quel est le format des paramètres envoyés et reçus par les clients et les serveurs ?
4. Quel algorithme d'ordonnancement des requêtes utilisé pour les processus serveurs ?

Les processus C/S utilisent des protocoles d'application pour envoyer et recevoir l'information. Dans la figure 3.2[1] ces protocoles sont implémentés dans la couche application suivant deux niveaux :

. Des protocoles de **bas niveau** (dépendant du réseau), exemple les sockets basés sur le protocole **TCP/IP**.
. Des protocoles de **haut niveau** comme le *Remote Procedure Call* (**RPC**).

3.3 Deux protocoles d'application

Dans le monde client/serveur on a deux protocoles :

Protocoles de base du Client/Serveur[2]

Plusieurs applications client/serveur scientifiques et d'ingénierie sont développées sous Unix et utilisent **TCP/IP** pour l'échange d'information entre processus s'exécutant sur différentes machines du réseau.

Les **Sockets Berkeley** (communément connu sous le nom de **Sockets**) sont fréquemment utilisés pour développer des systèmes client/serveur basés sur Unix-TCP/IP. Les Sockets Berkely sont des APIs et supportent deux domaines principaux : Le domaine Unix et le domaine Internet.

[1] [Uma93] p: 265
[2] [Uma93] p : 268-273

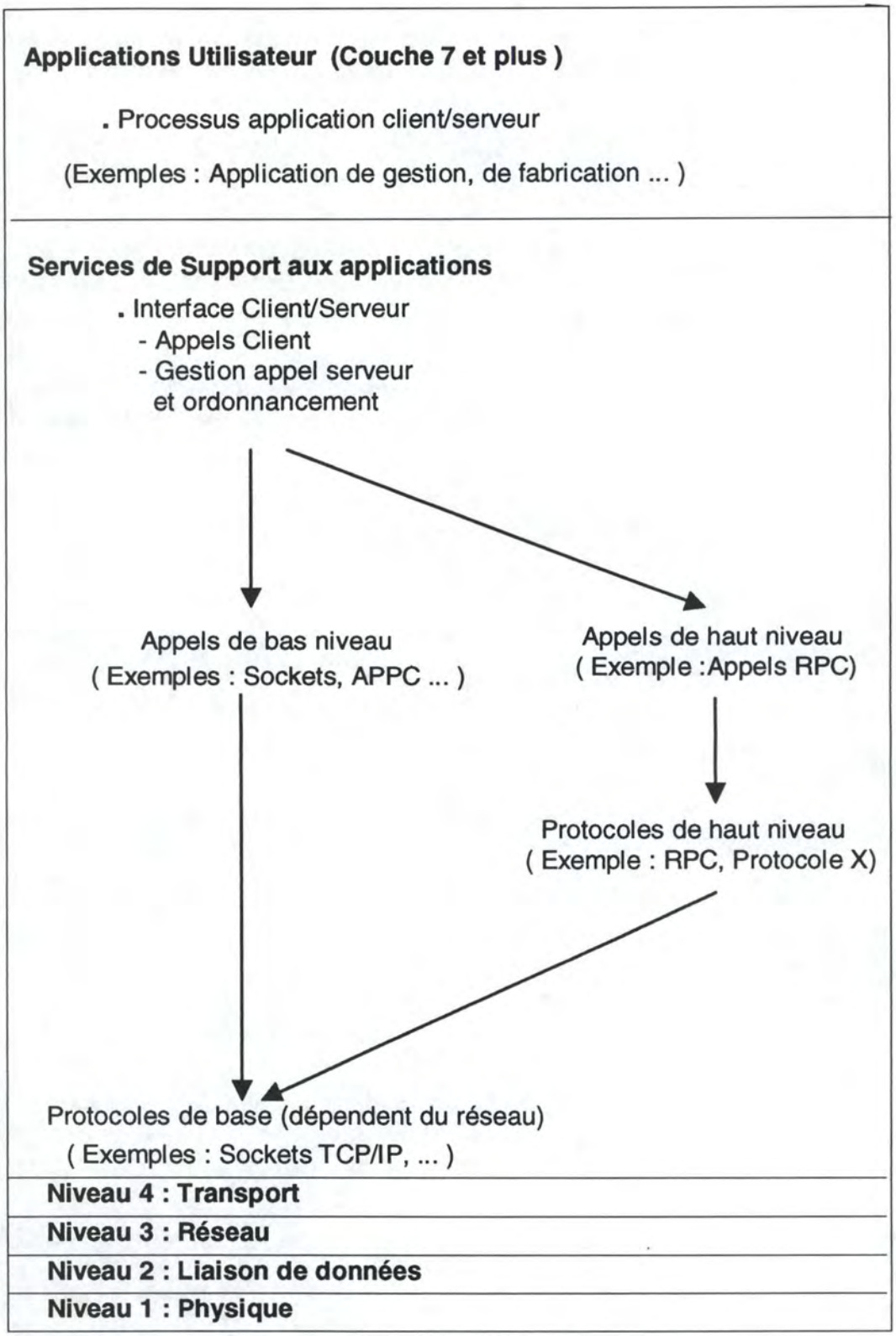


Figure 3.2 : Les services et les protocoles du client/serveur

Dans le monde **Internet**, un socket est un point de communication adressable qui conceptuellement se trouve au-dessus de la couche TCP. L'adresse associée à un socket est l'adresse physique IP (32 bits pour le **numéro du host**, 16 bits pour le **port**). Il y a différents types de sockets, regroupés suivant les services qu'ils fournissent. Ces services comportent des **séquences de sockets** qui fournissent un **service orienté connexion** (respect de la séquence, gestion des erreurs, ...) et des **sockets datagrammes** qui transfèrent des messages de différentes tailles dans les deux directions mais ne garantissent pas le respect de la séquence. Le tableau 3.1 regroupe les différentes commandes correspondantes à ces services.

Commande	Signification
SOCKET	Crée un socket et spécifie son type (TCP ou UDP)
BIND	Affecte un nom à un socket non nommé
CONNECT	Établit une connexion entre un host local vers un serveur distant
LISTEN	Le serveur formule son désir d'accepter des connexions
ACCEPT	Accepte une connexion et la mettre dans la file d'attente
WRITE	Envoie la donnée sur un socket TCP
READ	Lire la donnée du socket TCP
SENDTO	Envoie la donnée sur le socket UDP
RECVFROM	Lire la donnée du socket UDP

Tableau 3.1

Protocole de haut niveau : *Remote Procedure Call* (RPC)[1]

Le **RPC** permet à un appel de procédure au niveau du client d'être converti en un appel de procédure au niveau du serveur. Avec le RPC un processus local invoque des processus distants. Le RPC permet au programmeur de faire des appels de procédure de la même façon que les appels locaux. Le logiciel RPC cache aux developpeurs des applications Client/Serveur tous les détails concernant les réseaux. Par exemple un simple appel RPC ***open*** peut invoquer plusieurs commandes Socket comme **SOCKET**, **BIND**, **LISTEN**, **CONNECT** et **ACCEPT**.

On parle de **requête** pour désigner l'appel du client et de **réponse** pour désigner le résultat émis par le serveur. En général les RPC sont implémentés en dessus des sockets.

[1] [Uma93] p : 275-277

La figure 3.3[1] montre les étapes formant un RPC. Les routines client et serveur tournent dans deux processus séparés, et dans deux systèmes différents. Le logiciel RPC crée une procédure ***dummy*** avec le même nom que le serveur et la place dans le processus client. Cette procédure *dummy* appelée ***stub***, prend les paramètres appelés et les met dans des messages de transmission réseau adapté. Un autre ***stub*** est généré du côté serveur pour exécuter le traitement inverse.

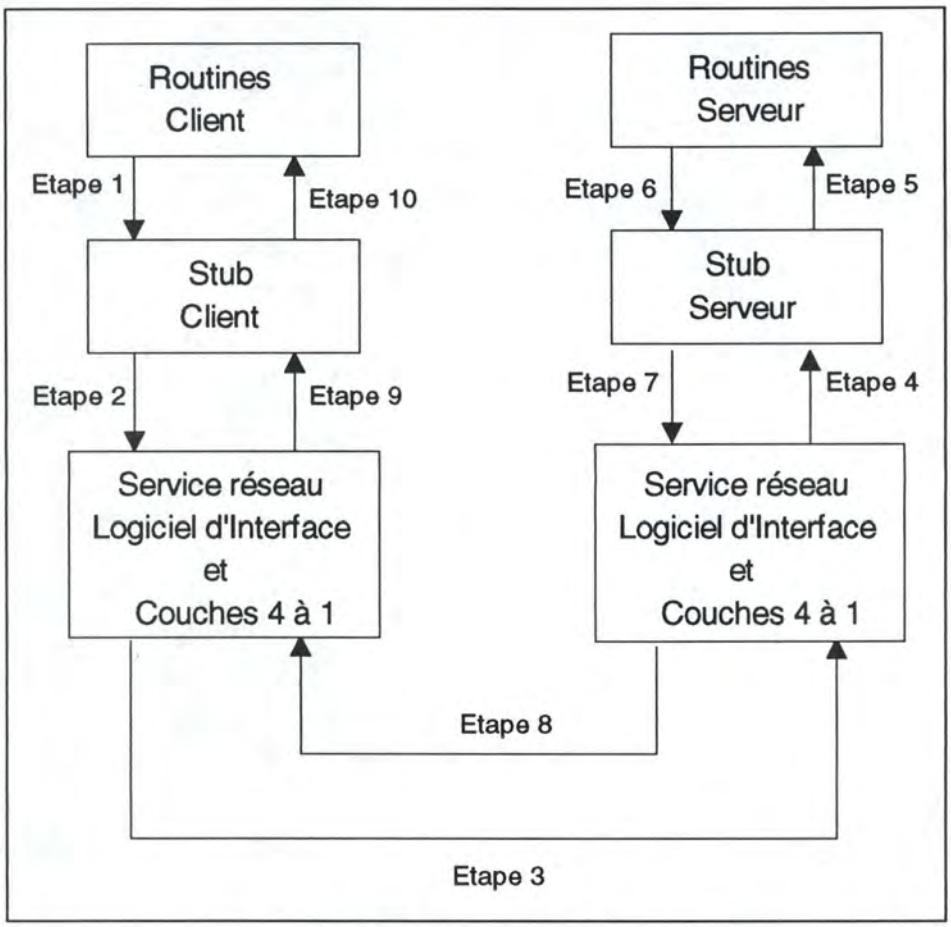


Figure 3.3 : Les étapes d'un RPC

Les étapes décrites dans la figure 3.3 sont exécutées dans l'ordre suivant :

Etape 1. Le client exécute un appel à une procédure locale, appelé un **client *stub***. Le client *stub* apparaît à l'appelant comme une procédure locale. Le *stub* traduit l'appel de procédure à distance en un message réseau approprié dans le format de Network Interface Services (NIS) tel Socket, APPC, MMS. Le *stub* détermine aussi l'adresse réseau du serveur.

[1][Uma93] p : 276

Etape 2. Les messages réseau sont envoyés à la couche transport du client.

Etape 3. Les messages sont envoyés via les moyens de communication en utilisant les couches inférieures (4 à 1) du client et ces messages sont reçus par le système réseau du serveur. Si l'architecture du réseau du client est différente de celle du serveur, un ***gateway*** est alors nécessaire (par exemple : *gateway* TCP/IP à SNA)

Etape 4. Le système réseau du serveur informe le *stub* du serveur qu'il a reçu une requête qui lui est adressé.

Etape 5. Le *stub* du serveur prend le message réseau, le traduit en un format d'un appel de procédure locale, et exécute cet appel pour le serveur.

Etape 6. Le serveur traite l'appel et développe une réponse, qui est envoyé au *stub* du serveur.

Etape 7. Le *stub* du serveur traduit la réponse en un ou plusieurs messages réseau, qui sont envoyés au système réseau. Le *stub* peut cacher plusieurs messages avant qu'une réponse appropriée ne soit développée au serveur.

Etape 8. Le système réseau du serveur envoie la réponse au système réseau du client en utilisant les couches inférieures (4 à 1). Un *gateway* peut être nécessaire si les architectures réseau du client et du serveur ne sont pas similaires.

Etape 9. Le système réseau du client envoie les messages de réponse au *stub* du client.

Etape 10. Le *stub* du client reçoit le message de réponse, le traduit et envoie la réponse au client. Le *stub* peut cacher plusieurs messages avant qu'une réponse appropriée ne soit développée au client.

3.4 Les paradigmes d'architecture client/serveur

La figure 3.4[1] montre le développement des architectures client/serveur. Ces applications ont évolué d'un modèle basé sur le système de fichier dans les années 80 en passant par des serveurs de base de données qui restent le modèle le plus utilisé. Avec le développement des technologies multimédia le modèle groupeware faisait son apparition au milieu des années 80.

[1] [Orf96] p : 20

Dans le monde mainframe on utilise le modèle des moniteurs transactionnels qui est bien approprié pour les applications sensibles *mission critical* nécessitant des traitements transactionnels. Actuellement on assiste à l'émergence du modèle client/serveur basé sur la technologie des objets distribués que l'on va présenter au chapitre 4.

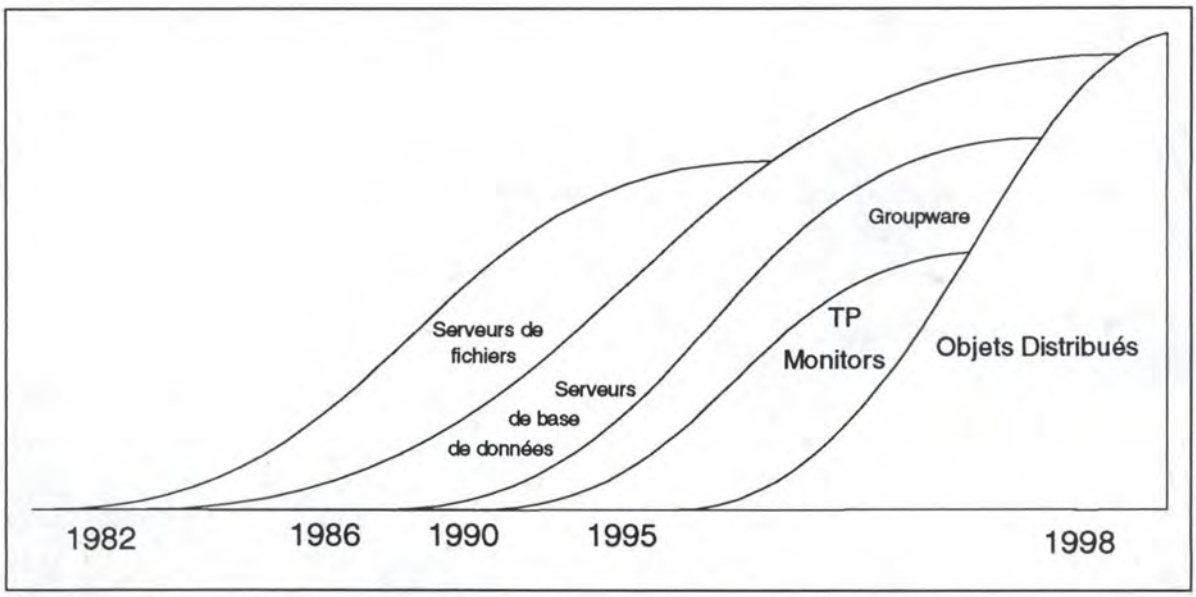


Figure 3.4 : Evolution des architectures client/serveur

Actuellement il y a 3 paradigmes[1] dans le monde client/serveur :

3.4.1 Client/Serveur avec base de données SQL

Le **serveur base de données SQL** est le modèle dominant pour créer des applications Client/Serveur. Mais seule la gestion des données n'est pas suffisante. Il faut aussi gérer les fonctions qui manipulent les données.

Avec le modèle basé sur SQL, on peut créer des applications client/serveur dans des environnements *single-vendor/single-server*. Dans ces environnements, on peut créer facilement des applications SQL en utilisant une panoplie d'outils *Graphical User Interface* (GUI).

[1] [Orf96] p : 7-13

Mais SQL présente les limitations suivantes :

. **SQL est pauvre dans la gestion des processus.** SQL gère les processus en utilisant des extensions non-standard de haut niveau et des langages procéduraux. En plus, il est difficile d'utiliser SQL pour exécuter des programmes qui doivent communiquer de façon transactionnelle à travers plusieurs serveurs.
. **Les Middlewares SQL ne sont pas standardisés.** SQL de différents vendeurs ne peuvent pas intéropérer. Il n'y a aucun protocole standard pour échanger les messages SQL entre différents vendeurs à travers le réseau.
. **SQL n'est pas approprié pour gérer des données riches.** SQL est basé sur des types simples de données, et donc n'est pas approprié pour gérer des types de données complexes.

3.4.2 Client/Serveur avec moniteurs TP (*transaction processing*)

Dans le monde mainframe, un **moniteur TP** (*Transaction Processing*) est toujours associé à chaque base de données. On ne peut pas créer des applications sans gérer les programmes (ou les processus) qui opèrent sur les données. Les moniteurs TP gèrent les processus et orchestrent les programmes. Ceci est fait en décomposant les applications complexes en **transactions** (une transaction s'exécute de façon atomique c'est-à-dire soit qu'elle s'exécute jusqu'au bout soit qu'elle ne s'exécute pas).

Le moniteur se positionne comme une couche indépendante des systèmes d'exploitation et des bases de données. Il agit comme une plate-forme prestataire de services vis-à-vis du client. A l'origine de cette évolution, le modèle XTP de l'X/Open qui garantit l'ouverture, la capacité du moniteur à supporter les systèmes distribués: le SGBDR, le gestionnaire de communications et l'application. A la différence des autres *middlewares*, le moniteur TP fournit des services de niveau application et orienté transaction. Outre la supervision et la synchronisation des transactions qui constituent leur mission d'origine et qui garantissent le niveau de performance et la sécurité des systèmes, les moniteurs remplissent des fonctions supplémentaires. Ils garantissent une optimisation de la charge entre les différents serveurs. Ils permettent aussi de fédérer des SGBD hétérogènes. Certains produits offrent une large palette d'outils. Quant aux modes de communication, les moniteurs apportent une grande flexibilité en supportant des fonctionnements en mode synchrone, conversationnel ou par files d'attente.

Les moniteurs transactionnels peuvent exécuter des classes d'application qui servent des milliers de clients. Elles fournissent un environnement qui se positionne entre les clients distants et les ressources du serveur. En se positionnant entre les clients et les serveurs, les moniteurs transactionnels peuvent gérer les transactions, les faire router à travers le système ou les réexécuter en cas de pannes. Les moniteurs transactionnels peuvent gérer les ressources sur un seul serveur ou à travers plusieurs serveurs et ils peuvent aussi coopérer avec d'autres moniteurs transactionnels dans des arrangements fédérés.

Les moniteurs transactionnels permettent aussi au système d'exploitation et aux gestionnaires de ressources au niveau du serveur de traiter avec un grand nombre de clients.

3.4.3 Client/Serveur avec *Groupware*

Le *groupware* est une collection de technologies représentant des processus complexes centrés autour des activités humaines, ces activités seront amenées à collaborer entre elles. Il est construit autour des technologies suivantes : la gestion des documents multimédia, le *workflow*, le courrier électronique, la gestion des conférences et la gestion d'agenda.

Le *groupware* collecte des données non structurées y compris textes, images, faxes, mails et des bulletins de bord et les organise en un nébuleux objet appelé : **document**. Les documents peuvent être visualisés, stockés, dupliqués et routés n'importe où sur le réseau. Les documents multimédia sont pour un *groupware* ce qu'est une table pour une base de données SQL. C'est l'unité de base à gérer.

3.5 Conclusion

Dans ce chapitre on a présenté un modèle générique d'architecture client/serveur. Par ailleurs on a fait un panorama des différents paradigmes qui existent à savoir client/serveur avec base de données SQL, client/serveur avec moniteurs TP et client/serveur avec *Groupware*. La figure 3.4 montre l'évolution des architectures client/serveur et on assiste actuellement à l'émergence du modèle client/serveur basé sur la technologie des objets distribués. Dans le chapitre suivant, ce modèle sera présenté, avec ce qu'il apporte de nouveau par rapport aux paradigmes présentés dans ce chapitre.

Chapitre 4 : Les objets distribués

4.1 Introduction

Dans les paradigmes client/serveur décrits dans le chapitre 3, on est passé des applications **centralisées monolithiques** à des applications découpées en deux parties. Ceci donne lieu à **deux applications monolithiques** : une sur le client et l'autre sur le serveur. Ces applications client/serveur restent donc difficiles à construire, à gérer et à étendre.

Les **objets distribués** permettent la subdivision des applications client/serveur monolithiques en composants **auto-gérables** qui collaborent ensemble et sillonnent à travers les réseaux et les systèmes d'exploitation.

La technologie des objets distribués est extrêmement appropriée pour créer des systèmes client/serveur flexibles puisque les données et la logique du traitement sont encapsulées dans les objets, leur permettant d'être localisés n'importe où dans un système distribué. La granularité de la distribution est hautement améliorée. Les objets distribués permettent des composants granulaires de logiciels capables de faire du ***plug-and-play*** (c'est-à-dire qu'à partir de composants existants on est capable de générer un autre composant que l'on met à la disposition des demandeurs de services), d'**interopérer** à travers le réseau, de **s'exécuter** dans différentes plates-formes et de **coexister** avec des applications *legacy*.

Les objets doivent être capables de gérer des systèmes complexes en diffusant des instructions et des alarmes. On peut changer n'importe quel objet sans affecter le reste des composants du système ou la façon avec laquelle ils interagissent.

Il y a toujours la question suivante qui se pose : Pourquoi cet intérêt renouvelé à la technologie objet qui existe il y a une vingtaine d'années? Actuellement, on peut dire que cette technologie est mûre et maîtrisée. L'industrie a créé aussi une référence pour une infrastructure des objets distribués qui inclut des bus logiciels d'objet et des technologies pour des composants capables de faire du *plug-and-play* dans ces bus. Les applications monolithiques sont construites en un seul bloc alors que les objets distribués et le bus d'objet permettront aux developpeurs de systèmes d'information d'opérer de façon incrémentale et de développer des applications à partir de composants existants.

Dans le reste du chapitre, on va étudier deux modèles basés sur la technologie objet à savoir : **CORBA (ORB)** et **OLE(COM/DCOM)**.

4.2 CORBA(ORB) [1]

4.2.1 Présentation générale

Depuis 1989, un consortium de vendeurs d'objets -*Object Management Group* **(OMG)**- a spécifié l'architecture d'un **bus logiciel ouvert** dans lequel des composants objet écrits par différentes sources peuvent interopérer à travers des **réseaux différents** et des **systèmes d'exploitation différents**. Actuellement OMG regroupe 500 compagnies et le **bus objet** est en train de devenir un pilier du *middleware* client/serveur.

Le bus objet fournit un ***Object Request Broker*** **(ORB)** qui permet aux clients d'invoquer des méthodes dans des objets distants aussi bien de façon statique que dynamique.

CORBA crée des **spécifications d'interface** et non **le code**. Les spécifications sont écrites dans un langage neutre l'***Interface Definition Langage*** (IDL) qui définie les frontières d'un composant. Les composants écrits en IDL sont neutres vis-à-vis des langages de programmation, des systèmes d'exploitation, et des réseaux.

Objets distribués CORBA

Les objets CORBA sont doués d'intelligence et peuvent vivre n'importe où sur le réseau. Le langage et le compilateur utilisés pour créer les objets serveurs sont totalement transparents aux clients. Les clients n'ont pas besoin de savoir où se trouvent les objets distribués et sur quel système d'exploitation ils vont s'exécuter. Les clients n'ont pas besoin de savoir comment les objets serveurs sont implémentés. Un objet serveur peut être implémenté en un ensemble de classes C++ ou en code COBOL, le client ne sent pas la différence. Par contre les clients doivent savoir **l'interface que leurs serveurs d'objet publient**. L'interface sert de **contrat liant le client au serveur**.

ORB et RPC

Qu'est ce qui fait la différence entre les invocations des méthodes ORB et ceux de RPC?

Les mécanismes sont similaires, mais il y a des différences importantes. Avec un RPC, on appelle une fonction spécifique, la donnée est séparée (figure 4.1). Mais sous ORB, on appelle une méthode dans un objet *spécifique*.

---

[1] [Orf96] p : 47-59

Différentes classes d'objet peuvent répondre différemment à la même invocation de méthode en utilisant le mécanisme du polymorphisme (figure 4.2).

Dans la figure 4.1 le client appelle une procédure et tout est connu lors de la **compilation** et il n'y a pas moyen de faire des **appels dynamiques** permettant de découvrir la procédure et ses paramètres au ***run-time***.

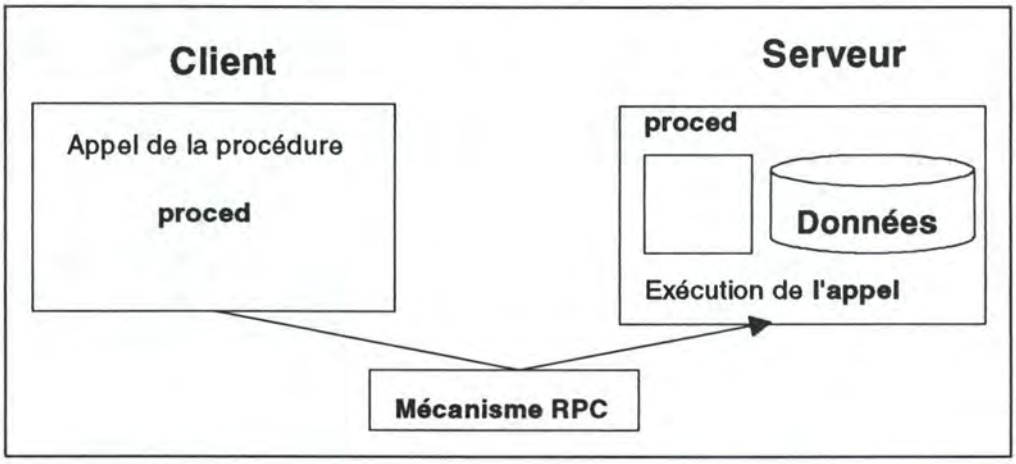


Figure 4.1 : Le mécanisme RPC

Alors que dans la figure 4.2 on invoque une méthode dans un objet. Cet objet encapsule aussi bien la méthode que la donnée. Grâce à un ***repository*** décrivant les objets, on peut faire de l'invocation dynamique à la différence de RPC où les appels sont statiques.

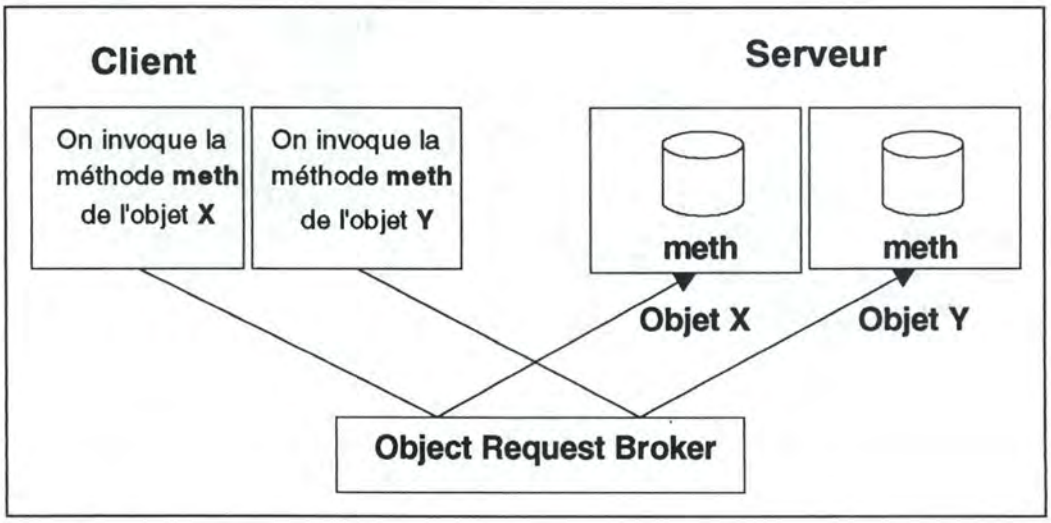


Figure 4.2 : Invocation d'une méthode à distance

4.2.2 CORBA 2.0 ORB[1]

La version CORBA 2.0 fournit un environnement pour gérer, informer les objets de la présence d'autres objets et décrire leurs méta-données. Avec CORBA 2.0 un ORB peut jouer le rôle de **courtier** entre deux objets résidant aussi bien à l'intérieur d'un seul processus (Un programme C++ par exemple) qu'entre deux objets qui interagissent à travers plusieurs ORBs et à travers plusieurs systèmes d'exploitation.

L'ORB CORBA est un **middleware** qui établit les relations client/serveur entre objets. En utilisant l'ORB, un objet client peut de façon transparente invoquer une méthode d'un objet sur le serveur, qui peut être sur la même machine ou sur le réseau. L'ORB intercepte l'appel et il est responsable de trouver un objet qui peut implémenter la requête, lui passe les paramètres, invoque ses méthodes, et retourne le résultat. Les objets sur l'ORB peuvent se comporter comme client ou serveur suivant les circonstances.

L'ORB permet donc aux objets d'émettre de façon transparente des **requêtes** et de recevoir des **réponses** d'autres objets localisés sur la même machine ou sur le réseau.

Le client n'est pas conscient des mécanismes utilisés pour communiquer avec les objets du serveur.

L'anatomie de l'ORB CORBA 2.0

La figure 4.3 montre les faces client et serveur d'un ORB CORBA.

• **Du côté client:**

. **Les stubs IDL client (IDL *Stub*).** Ces stubs fournissent des **interfaces statiques** aux services objet. Ces stubs précompilés définissent comment les clients invoquent les services correspondants sur les serveurs. Du côté du client, le *stub* se comporte de la même façon qu'un appel local, c'est un **mandataire** local pour un objet serveur distant.
Les services sont définis en utilisant l'IDL, et les deux *stubs* client et serveur sont générés par le compilateur IDL. Le client doit avoir un IDL *stub* pour chaque interface utilisée sur le serveur. Le *stub* contient du code pour exécuter toutes les opérations de ***marshaling*** c'est-à-dire il encode (et décode) les opérations et leurs paramètres dans des formats de message qui peuvent être envoyés au serveur.

---

[1] [Orf96] p : 67-79

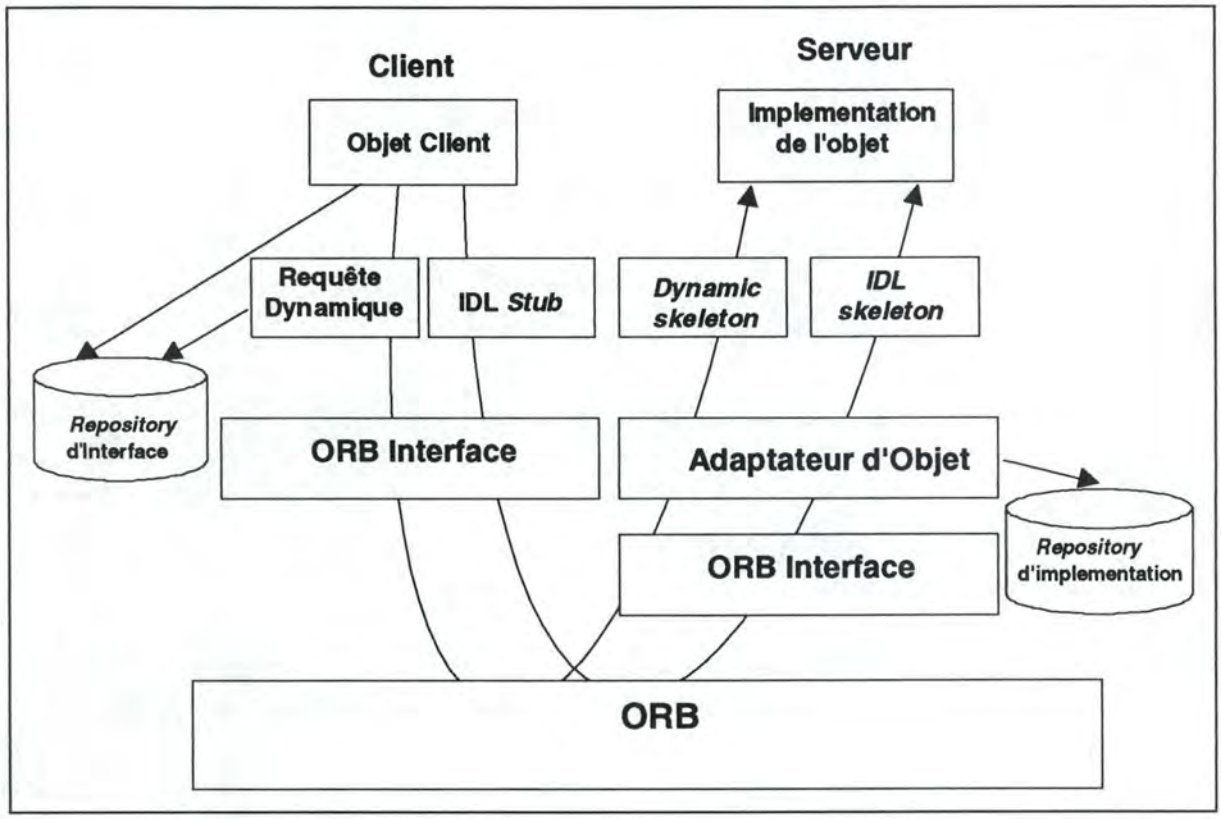


Figure 4.3 : Les composants de CORBA

Il contient aussi des en-têtes de fichier qui permettent au client d'invoquer des méthodes sur le serveur à partir des langages de haut niveau (comme C, C++ ou Smaltalk) sans se soucier des protocoles sous-jacents. Le client invoque la méthode à partir de son programme pour obtenir le service distant.

. **Requête Dynamique.** Une requête dynamique permet au client de découvrir la méthode à invoquer au *run-time*. CORBA défini des API standards pour faire des recherches dans les méta-données qui définissent l'interface serveur, génèrent les paramètres, lancent l'appel distant et récupèrent le résultat en retour.

. **Les API du repository d'interface (Interface *Repository*).** Ces API permettent au client d'obtenir et de modifier les descriptions des toutes les interfaces des composants enregistrés, les méthodes qu'elles supportent, les paramètres et ce au ***run-time***. En CORBA, ces descriptions sont appelées **signatures des méthodes**.

On peut voir le *repository* d'interface comme un *repository* de méta-données dynamique pour les ORB. Les API permettent aux composants de façon dynamique (c-à-d au *run-time*) d'accéder, de stocker et de mettre-à-jour les informations sur les méta-données. Cette utilisation des méta-données permet à chaque composant vivant dans l'ORB d'avoir des interfaces auto-déscriptives. L'ORB est lui-même un bus **auto-déscriptif**.

. L'interface ORB (ORB Interface). Cette interface est une API qui permet l'accès à des services locaux qui peuvent être d'un intérêt pour une application donnée. Par exemple, CORBA fournit des API pour convertir une référence à un objet en un string et vice-versa. Ces appels peuvent être utiles si on a besoin de stocker ou de communiquer des références d'objet.

Le support des invocations client/serveur statiques et dynamiques, aussi bien le *repository* d'interface, donnent à CORBA un grand avantage par rapport aux autres *middleware*.
Les invocations statiques sont faciles à programmer et sont auto-documentées. Les invocations dynamiques fournissent un maximum de flexibilité, mais elles sont difficiles à programmer, elles sont très utiles pour les outils qui utilisent des services au *run-time*.

• **Du côté serveur :**
Du côté serveur, on ne fait pas la différence entre les invocations statiques et dynamiques, elles ont toutes les deux la même sémantique de message. Dans les deux cas l'ORB localise un ***object adapter***, transmet les paramètres, et transfère le contrôle à l'implémentation de l'objet à travers le *stub* IDL serveur (dans le monde CORBA on parle de ***skeleton*** pour désigner les *stubs* du côté du serveur). Du côté du serveur on trouve :

. Les *stubs* IDL serveur (IDL *skeleton*). Ces *stubs* fournissent des interfaces statiques à chaque service exporté par le serveur. Ces *stubs*, comme ceux du côté client, sont créés en utilisant le compilateur IDL.

. *Dynamic Skeleton* . Les *Dynamic Skeletons* prennent les valeurs des paramètres d'un message entrant afin de déterminer pour qui est le message c'est-à-dire l'objet et la méthode cible à la différence des *skeletons* compilés qui sont définies pour une classe d'objet particulière et nécessite une implémentation pour chaque méthode IDL définie. Les *Dynamic Skeletons* sont très utiles pour implémenter des bridges génériques entre ORB.

Elles peuvent aussi être utilisées par les interpréteurs et par des langages de *scripting* pour générer de façon dynamique des implémentations d'objet. Le *dynamic skeleton* est l'équivalent serveur de la **Requête Dynamique**. Il peut recevoir aussi bien des invocations statiques que dynamiques.

. Adaptateur d'Objet (*Object Adapter*). L'adaptateur d'objet se positionne au sommet du noyau de services de communication de l'ORB et accepte des requêtes de service au nom des objets serveur. Il fournit l'environnement d'exécution pour instancier les objets du serveur, leur passer les requêtes, et leur assigner des identificateurs d'objet (IDs), CORBA les appelle des **références d'objet**. L'adaptateur d'objet enregistre aussi les classes qu'il supporte et leurs instances au *run-time* dans le **repository d'implémentation**.

. Le repository d'implémentation (*Implementation Repository*). Il fournit un repository d'information au *run-time*. Ces informations portent sur les classes que le serveur supporte, les objets qui sont instanciés, et leurs IDs. Il sert aussi de place commune pour stocker des informations additionnelles associées avec l'implémentation des ORB.

. L'interface ORB (*ORB Interface*). Cette interface consiste en API pour accéder à des services locaux qui sont identiques à ceux fournit du côté client.

Invocation de méthodes statiques : Du IDL vers les *stubs* d'interface

La figure 4.4[1] montre les étapes à suivre pour créer des classes serveur, leur fournir des interfaces *stub*, stocker leurs définitions dans le *repository* d'interface, instancier l'objet lors de l'exécution, et enregistrer leur présence avec le *repository* d'implémentation.

Voici-ci contre les étapes :

Etape 1. Définition des classes d'objet en utilisant l'*Interface Definition Language* **(IDL)**.
L'IDL est le moyen par lequel les objets informent leurs clients potentiels des opérations disponibles et comment elles doivent être invoquées. Le langage de définition IDL définit les types d'objet, leurs attributs, les méthodes qu'ils exportent, et les paramètres des méthodes. L'IDL CORBA est un sous-ensemble d'ANSI C++ avec des constructions additionnelles pour supporter la distribution.
L'IDL est purement un langage déclaratif. Il utilise la syntaxe C++ pour les constantes, les types et les définitions des opérations mais n'incluent pas de structures de contrôle ou de variables.

---

[1] [Orf96] p : 75

Etape 2. Exécution du fichier IDL par un précompilateur de langage
Un précompilateur *CORBA-compliant* typique traite les fichiers IDL et produit des *skeletons* de langage pour l'implémentation des classes du serveur.

Etape 3. Ajout du code d'implémentation aux *skeletons*
On doit fournir le code qui implémente les méthodes dans les skeletons.

Etape 4. Compilation du code
Un compilateur *CORBA-compliant* est capable de générer au moins les 4 fichiers suivants:

. ***import files*** qui décrivent les objets.
. ***client stubs*** pour les méthodes définies dans la déclaration IDL, ces *stubs* sont invoqués par un programme client qui veut accéder de façon statique aux services définis dans la déclaration IDL via l'ORB.
. ***server stubs*** qui invoquent les méthodes sur le serveur.
. le **code** qui implémente les classes serveur.

La génération automatique des stubs libère les developpeurs de la nécessité de les écrire, et libère aussi les applications des dépendances par rapport à un ORB particulier.

Etape 5. Compilation des définitions des classes
Un outil est utilisé pour compiler les informations contenues dans la déclaration IDL dans le *repository* d'interface. Les programmes peuvent accéder à ces interfaces au *run-time*.

Etape 6. Instanciation des objets sur le serveur
L'adaptateur d'objet peut instancier des objets serveur qui vont servir les appels de méthodes des clients à distance.

Etape 7. Enregistrement des objets utilisés au *run-time* dans le *repository* d'implémentation
L'adaptateur d'objet enregistre dans le repository d'implémentation la référence à l'objet et le type de n'importe quel objet qu'il instancie dans le serveur. Le repository d'implémentation connaît aussi quelles sont les classes d'objet qui sont supportées par le serveur.

L'ORB utilise cette information pour localiser les objets actifs ou pour activer des objets dans un serveur particulier.

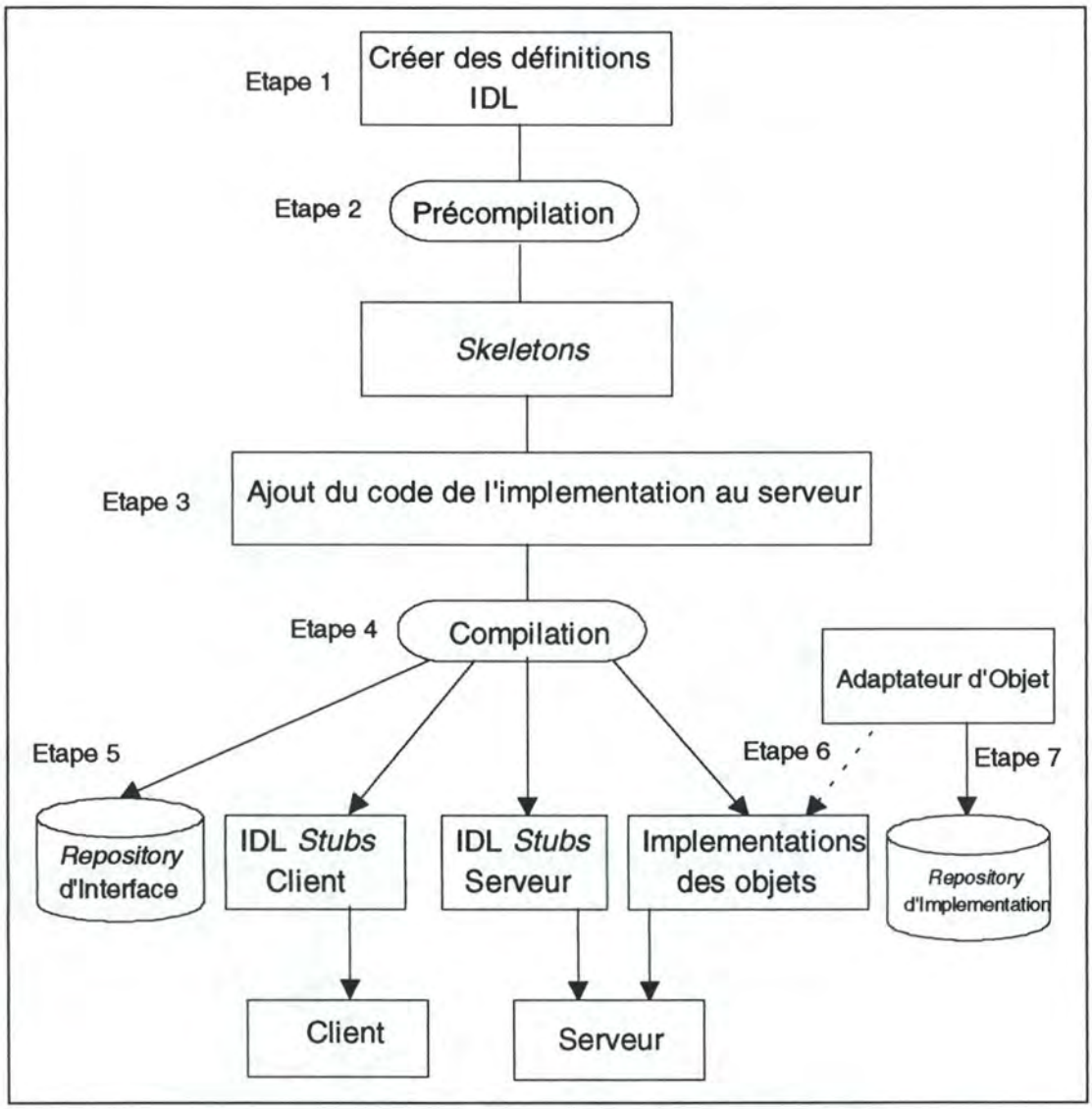


Figure 4.4 : Création d'objets serveurs

Invocation dynamique des méthodes

L'invocation dynamique permet à un programme client de construire et d'invoquer de façon dynamique des requêtes aux objets. Le client spécifie l'objet, la méthode à invoquer, ainsi que l'ensemble des paramètres à travers un ou plusieurs appels. Le client obtient ces informations en consultant le *repository* d'interface ou une source similaire et ceci se fait au ***run-time***.

L'invocation permet un maximum de flexibilité en permettant à de nouveaux objets d'être ajoutés au système distribué en temps d'exécution.

Voici les étapes à accomplir pour faire de l'invocation dynamique :

Etape 1. Obtention de la description de la méthode du *repository* d'interface.
CORBA spécifie quelques 10 méthodes pour localiser et décrire les objets dans le repository. Après la localisation de l'objet, on exécute l'appel *describe* pour obtenir la définition complète de l'IDL de l'objet.

Etape 2. Création de la liste des arguments
CORBA spécifie une auto-définition des structures de données pour le passage des paramètres. La liste des arguments est créée en utilisant l'opération *create_list* et on exécute l'appel add_arg autant de fois jusqu'à l'ajout de tous les arguments dans la liste.

Etape 3. Création de la requête
La requête doit spécifier la référence à l'objet, le nom de la méthode, et la liste des arguments. La requête est créée par l'appel *create_request*.

Etape 4. Invocation de la requête

On peut invoquer la méthode selon trois façons :

. L'appel *invoke* envoie la requête et obtient les résultats
. L'appel *send* retourne le contrôle au programme, qui doit lancer un *get_response* ou un *get_next_response*
. L'appel *send* est un appel à une direction, dans ce cas aucune réponse n'est requise

Rôle de l'Adaptateur d'Objet (Object Adapter)

L'Adaptateur d'Objet (figure 4.5) permet à une implémentation d'objet d'accéder aux services ORB.

Il fournit un environnement total pour l'exécution de l'application serveur. Voici quelques services fournis par l'Adaptateur d'Objet (figure 4.5) :

1. **Enregistrement des classes serveur dans le *repository* d'implémentation.**
On peut voir le repository d'implémentation comme un stockage persistent d'objet que l'adaptateur d'objet devra gérer. Les implémentations des objets sont enregistrées et stockées dans le repository d'implémentation.

2. **Instanciation de nouveaux objets au *run-time.***
L'Adaptateur d'Objet est responsable de la création des instances d'objet. Le nombre d'instances créées dépend de la charge du trafic entrant du client. L'Adaptateur est responsable du maintien de l'équilibre entre la fourniture des objets et les demandes des clients entrants.

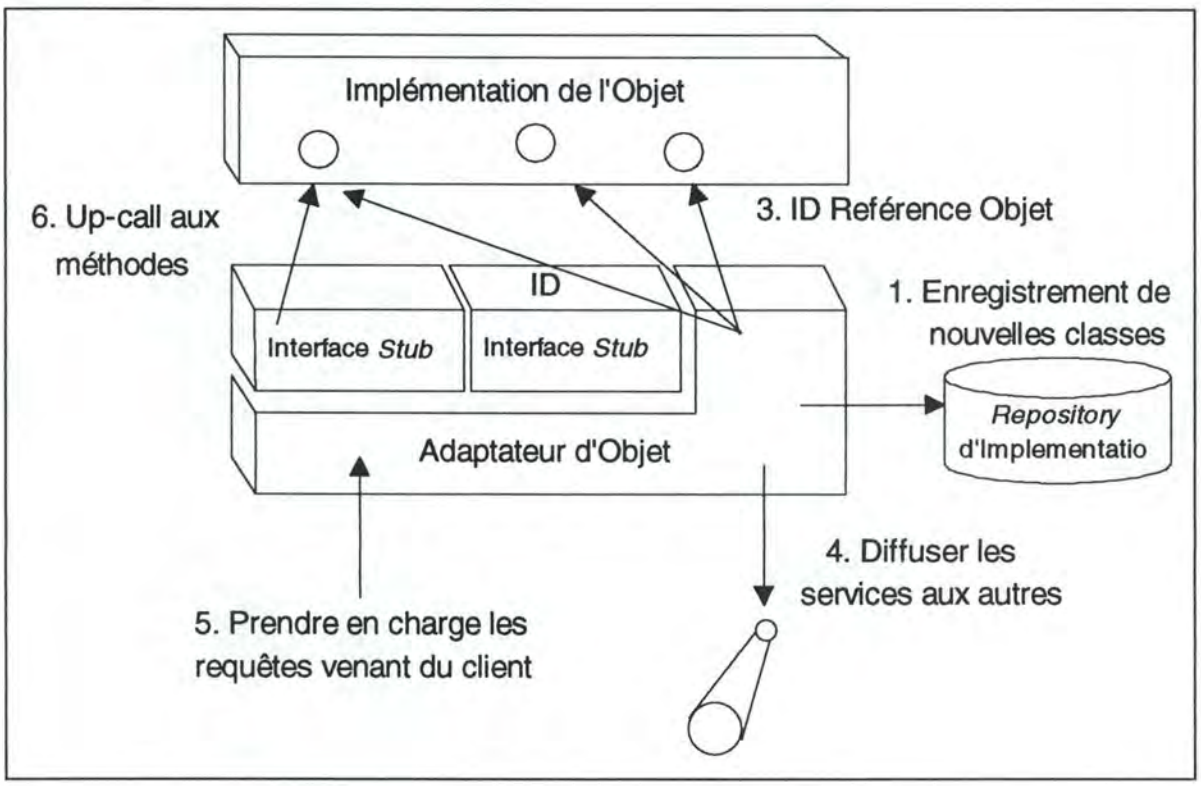


Figure 4.5 : Fonctionnement de l'Adaptateur d'Objet

3. **Génération et gestion des références objets.**
L'Adaptateur d'Objet affecte des références (des ID uniques) aux nouveaux objets crées.

4. **Signalement de la présence des serveurs objet.**
L'Adaptateur d'Objet peut signaler la présence des services qu'il fournit sur le bus ORB.

5. **Gestion des appels clients entrants.**
L'Adaptateur d'Objet interagit avec la couche supérieure du noyau. Le *stub* est responsable de l'interprétation des paramètres entrants et de leurs présentations dans une forme acceptable aux invocations de méthode des objets.

6. **Routage des appels entrants.**
L'Adaptateur d'Objet est implicitement impliqué dans l'invocation des méthodes décrites dans les *skeletons*. Par exemple, l'Adaptateur d'Objet peut activer l'implémentation. Il peut aussi authentifier les requêtes entrantes.

IDL et Le *repository* d'interface

L'objectif de CORBA est que la couche *middlware* d'un système client/serveur et tous les objets vivant dans l'ORB reposent sur le langage d'interface IDL. OMG permet d'atteindre ce but en deux étapes :

1. L'**IDL CORBA** permet aux fournisseurs de composants de spécifier l'interface et la structure des objets qu'ils fournissent dans un langage de définition standard. Un contrat IDL lie les fournisseurs des objets distribués à leurs clients. Pour qu'un objet puisse demander un service d'un objet, il doit nécessairement connaître l'interface de l'objet cible. Le *repository* d'interface CORBA contient les définitions de toutes ces interfaces. Il contient les metadonnées qui permettent aux composants de se reconnaître dynamiquement au *run-time*. Ceci permet à CORBA d'être **auto-déscriptif**.

2. On a aussi un ensemble de services distribués que les fournisseurs mettent à la disposition des objets clients. Ces services déterminent quels sont les objets présents sur le réseau et quelles méthodes ils fournissent. La localisation de l'objet doit être transparente au client.

Les méta-données permettent la création de systèmes client/serveur dits **agiles**. Un système agile est un système auto-déscriptif, dynamique et reconfigurable. Le système permet aux composants de se découvrir les uns les autres au *run-time*, il fournit l'information leur permettant d'interopérer.

Un système agile se démarque d'un système client/serveur traditionnel par son utilisation de méta-données pour décrire de façon cohérente tous les services disponibles, les composants et la donnée. Les méta-données permettent aux composants de découvrir de façon dynamique l'existence d'autres composants et de collaborer entre eux.

CORBA est un système agile. Un composant est conforme au modèle CORBA s'il est auto-déscriptif. Chaque objet système et chaque service vivant sur le bus CORBA doit être auto-déscriptif.

Le *repository* d'interface est une base de données que l'on peut consulter au *run-time* et qui contient les spécifications des interfaces de chaque objet présent sur l'ORB.

Gestion des objets Persistants [1]

A la différence des objets d'un programme, les objets distribués sont **persistants**. Pour cela, ces objets doivent être stockés dans des fichiers ou dans des bases de données. Une question survient : qui contrôle ces objets persistants?

C'est le *Persistent Object Service* (POS) de CORBA qui gère la persistance des objets distribués.

• ***Persistent Object Service*** **(POS figure 4.6)**

POS permet aux objets de persister au-delà de l'application qui les a crée ou du client qui les utilise. POS stocke l'objet et le restaure quand c'est nécessaire.

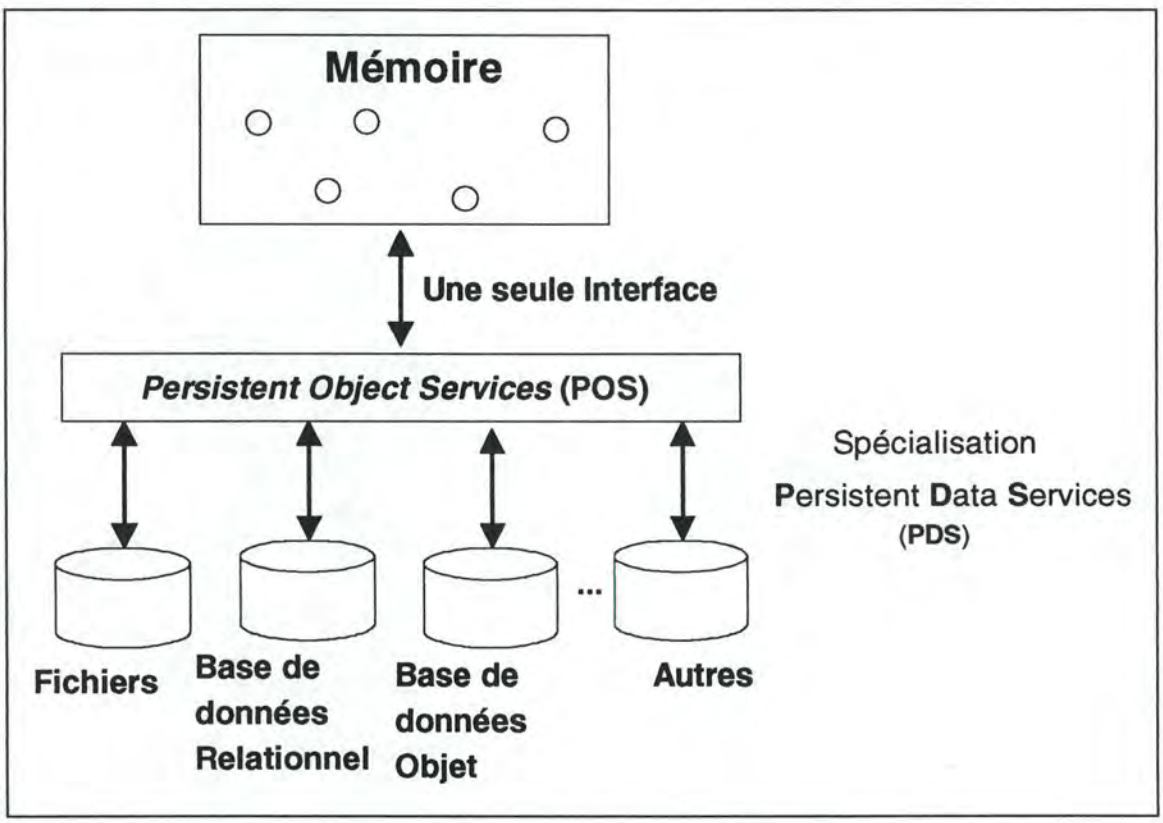


Figure 4.6 : Les services des objets persistants

Le but est de créer une implémentation ouverte qui rempli différents besoins de stockage persistants. Donc l'idée principale derrière POS est qu'une **seule interface objet** soit disponible pour de **multiple stockage de données**.

[1] [Orf96] p : 139-146

• **Stockage à un niveau et à deux niveaux.**

POS fournit une seule interface client pour stocker les objets sans se soucier de leurs mécanismes de stockage. POS peut gérer aussi bien le stockage à un seul niveau (par exemple, les SGBDOO) que le stockage à deux niveaux (par exemple, les base de données relationnelles ou simplement les systèmes de fichiers). Dans le stockage à un seul niveau, le client ne se soucie pas si l'objet est en mémoire ou en disque. Par contre dans le stockage à deux niveaux l'objet doit être explicitement chargé de la base de données (ou le fichier) vers la mémoire centrale et vice-versa.

• **POS : la vue du client.**

Parfois les clients d'un objet ont besoin de contrôler ou d'assister la gestion de la persistance. POS peut accommoder l'implication du client à différents niveaux. A l'extrême, le service peut être totalement transparent aux applications clientes.
Ceci veut dire que le client ignore le mécanisme de persistance, les objets apparaissent sur demande peu importe l'état dans lequel ils se trouvent.

A l'autre extrême, les applications clientes peuvent utiliser des protocoles de stockage spécifique pour "remonter en surface" tous les détails du mécanisme de stockage persistant sous-jacent.
Le client peut choisir quelle gestion de données persistants lui convient. Il faut noter que ceci conduit à un mécanisme à deux sens : Les objets persistants peuvent choisir de ne pas exporter leurs persistances aux clients. POS fournit des opérations sur des interfaces que les clients utilisent pour contrôler la persistance. Ces interfaces n'abandonnent pas le principe de l'encapsulation des données, mais ils fournissent aux clients une certaine visibilité. Plus encore, ils permettent aux clients de décider quand les objets persistants sont stockés et quand ils sont restaurés.

• **POS : la vue de l'*objet persistent* (OP).**

L'*objet persistant* est en charge de sa persistance, il décide quel protocole de stockage de données il faut utiliser et quel degré de visibilité il faut donner aux clients. L'OP peut aussi déléguer la gestion de ses données persistantes aux services de persistance sous-jacents. Mais, il peut aussi maintenir le contrôle sur les interactions avec le système de stockage.

Le PO peut aussi hériter la plupart des fonctions dont il a besoin pour maintenir sa persistance. Au minimum, un OP doit collaborer avec son stockage de données pour traduire ses états afin de permettre au service sous-jacent de prendre en charge une partie.

• **Les composants d'un POS.**

Un POS est constitué des éléments suivants (figure 4.7) :

. *Objets Persistants* **(OPs)**. Les objets persistants sont des objets dont l'état est stocké de façon persistante. Un objet peut être persistent en héritant (via IDL) de la classe **prédéfinie PO**. Il doit hériter (ou fournir) un mécanisme pour extérioriser ses états quand le mécanisme de stockage sous-jacent le lui demande (via un protocole). Chaque objet persistant possède un identificateur de persistance (PID) qui décrit la localisation à l'intérieur du stockage de données de cet objet en utilisant un identificateur *string*. Les clients interagissent avec l'interface de l'OP pour contrôler la persistance des objets.

. *Persistent Object Management* **(POM)**. Le gestionnaire des objets persistants (POM) est une interface indépendante pour les opérations de persistance. Il isole les OPs des services de données persistants particuliers. Le POM permet le routage des appels de l'objet persistant vers le service de données persistantes approprié en analysant l'information contenue dans le PID. Un POS possède un seul POM, placé entre les objets et les services de données persistants.
Le POM est un routeur pour le stockage des données. Il fournit une vue uniforme de la persistance dans le système à travers de multiples services de données.

. *Persistant Data Services* **(PDSs)**. Les services de données persistantes sont des interfaces vers des implémentations particulières de stockage de données. Les PDSs permettent aux données de "bouger" entre l'objet et le stockage de données. Les PDSs doivent obligatoirement implémenter l'**interface IDL prédéfinie : PDS**. En plus, des PDSs peuvent supporter des protocoles dépendants.

. **Les stockages de données**. Les stockages de données sont des implémentations qui stockent les données persistantes des objets indépendamment de l'espace d'adressage contenant l'objet. On peut avoir des SGBDOOs, des bases de données relationnelles ou des systèmes de fichier.

Pour la plupart des applications clientes, le mécanisme de persistance est totalement transparent. Si l'utilisateur désire contrôler la persistance des objets, il peut utiliser l'interface OP. Évidemment, l'objet est responsable du stockage et de la restauration de son état. L'utilisateur peut seulement dire quand il faut agir de la sorte en utilisant l'interface OP.

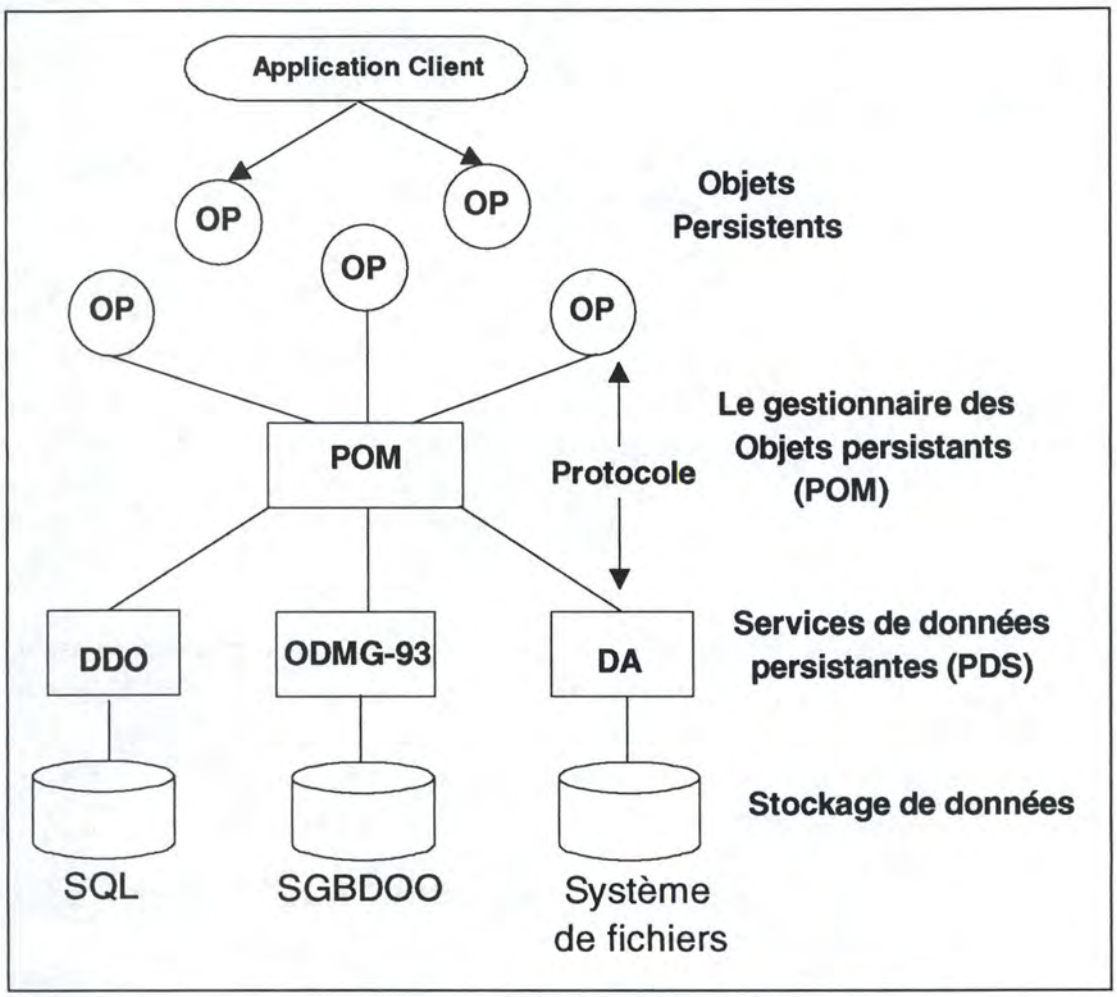


Figure 4.7 : Le gestionnaire des objets persistants

4.3 OLE/COM

4.3.1 Présentation de OLE/COM

Au départ l'*Object Linking and Embedding* (**OLE**) désignait le concept de document composé OLE (***OLE compound document***) (par exemple un tableau Excel incorporé dans un document Word). Actuellement c'est un standard pour la communication entre différentes applications. Comme CORBA, OLE offre des services communs permettant aux composants de collaborer de façon intelligente.

Le *Component Object Model* (**COM**) est un ORB pour une seule machine. Dans le paragraphe 4.3.4 on va voir DCOM qui est l'équivalent de COM sur un système distribué.

4.3.2 COM [1]

On va étudier COM à travers ses **composants** et ses **fonctions**.

Les composants de COM

Les composants de COM sont :

• **L'Interface COM**

Une **interface COM** est une collection d'appels de fonctions appelés aussi méthodes. Comme pour CORBA, une interface COM est un contrat client/serveur. Il définit les fonctions indépendamment de leurs implémentations. Les clients COM interagissent entre eux et avec le système en appelant les fonctions membres de l'interface. Comme pour CORBA, une interface COM est indépendante d'un langage particulier. Par contre à la différence de CORBA, COM ne fournit pas de *binding* de langage de haut niveau. COM ne définit pas de *binding* pour Smaltalk, C++, C et COBOL qui permet au code source d'être portable. COM définit simplement une spécification binaire sur comment accéder à ces interfaces en utilisant les pointeurs.

Pour accéder à une interface, les clients COM utilisent des pointeurs à un tableau de pointeurs de fonctions connus sous le nom de ***virtual table*** (*vtable*). Les fonctions pointées par le *vtable* sont des implémentations des méthodes des objets serveurs. Chaque objet COM possède un ou plusieurs *vtables* qui définissent le contrat entre l'implémentation de l'objet et ses clients.

• **L'Objet COM**

Un objet COM est un composant qui supporte une ou plusieurs interfaces comme définie par cette classe d'objet. Une interface COM se réfère au groupe de fonctions prédéfinies. Une classe COM implémente une ou plusieurs interfaces et elle est identifiée par un identificateur de classe unique **CLSID**. Un objet particulier fournit des implémentations des fonctions pour toutes les interfaces que sa classe supporte.
Les clients manipulent un objet COM via des pointeurs d'interface, ils n'accèdent jamais directement à l'objet lui-même. L'interface dans le monde COM n'est pas un objet au sens classique. Les objets COM ne supportent pas un identificateur objet unique comme pour CORBA (via les références objet).

Tous les objets COM implémentent l'interface **IUnkown** à travers laquelle le client peut contrôler le cycle de vie de l'objet. Les clients utilisent cette interface particulière pour négocier et peuvent demander à un objet quelles interfaces il supporte et obtenir des pointeurs sur elles.

---

[1] [Orf96] p : 429-437

• **Le Serveur COM**

Un serveur COM est une pièce de code (une DLL ou une EXE), qui loge une ou plusieurs classes d'objet chacune avec son propre CLSID. Quand un client demande un objet d'un certain CLSID, COM charge le code serveur et lui demande de créer un objet de cette classe.

Le serveur doit fournir une "usine" de classe *class factory* pour créer un nouvel objet. Une fois que l'objet est créé, un pointeur vers son interface primaire est renvoyé au client.

Un serveur COM fournit la structure nécessaire pour qu'un objet soit disponible aux clients. Plus précisément un serveur COM doit :

. **Implémenter une interface de l'usine de classe**. Le serveur doit implémenter une usine de classe avec l'interface ***IClassFactory*** pour chaque CLSID supporté. L'usine de classe crée des instances d'une classe.

. **Enregistrer les classes qu'il supporte**. Le serveur doit enregistrer le CLSID de chaque classe qu'il supporte avec le ***Window Registry***. Pour chaque CLSID, il doit créer un ou plusieurs entrées qui fournissent des noms de chemins au serveur DLL ou EXE. Cette information est enregistrée en utilisant les API *Window Registry*. Typiquement les classes sont enregistrées au moment de l'installation.

. **Initialiser la librairie COM**. Le serveur lance un appel à l'API COM ***CoInitialize*** pour initialiser COM. Il fournit les services et les API COM au *run-time*.

. **Vérifier que la librairie est une version compatible**. Le serveur exécute ceci en appelant l'API ***CoBuildVersion***.

. **Fournir un mécanisme de terminaison**. Le serveur doit fournir une méthode pour terminer son exécution quand il n'y a aucun client actif pour ses objets.

. **Libérer la librairie COM**. Le serveur appelle l'API COM ***CoUninitialize*** quand il n'est plus utilisé.

Les fonctions de COM[1]

COM fournit un nombre de routines utiles pour les developpeurs de logiciels. En général ces fonctions commencent par le mot **Co** et possède des noms comme **CoInitilialize** et **CoCreateInstance**.

On retrouve les fonctions suivantes :

• ***Marshaling***
COM est responsable de "la mise en paquet" (***packaging***), de l'envoie et de "l'extraction du paquet" (***unpackaging***) des paramètres d'interface à travers les frontières du processus, de la machine et du réseau. Le *marshaling* détermine comment les paramètres sont passés, les principes de codage et comment ces paramètres sont identifiés dans le message.

Actuellement, le mécanisme de transport est fourni par le système d'exploitation lui-même et il n'est pas considéré comme faisant partie de COM. Localement, COM utilise un appel de procédure appelé ***Lightweight Remote Procédure Call*** (LRPC); et à distance il utilise le standard industriel DCE RPC.

• **Stockage structuré**
COM fournit un système complet pour gérer le stockage et les séquences d'objets de façon robuste, persistante et hiérarchique. En général, un objet stocké structuré se comporte comme un volume disque, en effet il peut traduire les contenus (comme une table d'allocation de fichier), un ou plusieurs objets stockés (analogue aux répertoires racines et les sous-répértoires), et un ou plusieurs séquences d'objets (similaire aux fichiers dans les répertoires). Les objets stockés structurés peuvent être rassemblés et emboîtés, et peuvent exister dans un fichier disque, en mémoire, ou comme des enregistrements de base de données.
En plus de ces caractéristiques, le stockage structuré fournit aussi la notion de **transaction**, que l'on peut utiliser pour implémenter l'opération ***Undo***. OLE fournit aussi une implémentation par défaut de stockage structuré appelé les fichiers composés, duquel les *OLE compound documents* sont dérivés.

• ***Monikers***
Un *moniker* représente le nom d'un objet COM spécifique. Un *moniker* contient de l'information sur un objet et les instructions pour se connecter à cet objet.

---

[1] [Ple96]

• **Uniform data transfer**

Uniform Data Transfer (UDT) est un mécanisme important dans n'importe quelle application qui manipule des composants. COM garanti que les services OLE utilisant par exemple le bloc-notes, les opérations de *drap-and-drop*, les ***OLE automation*** utilisent tous des formats de données compatibles.

• **Gestion des versions**

L'utilisation de l'interface COM crée un contrat entre le fournisseur de l'objet et le consommateur.

Il est important que ce contrat ne soit pas cassé si l'objet évolue. La gestion de la version de l'interface COM permet l'ajout de services aux objets sans affecter les applications existantes.

4.3.3 OLE[1]

OLE est un ensemble de services objet au dessus de COM. Le premier service offert est la notion ***OLE compound documents***. On est passé après vers ***OLE automation*** qui était disponible seulement via Visual Basic et enfin vers ***OLE controls*** qui est un hybride de *OLE compound documents* et d'*OLE automation*.

Dans la suite on définit ces différents concepts :

• ***OLE compound documents***

Les *OLE compound documents* sont des formes de documents composés qui incorporent des données créées par différents applications OLE. Un exemple est celui d'un objet tableau **Excel** inclus dans un document **Word**.

Les programmes application qui créent les documents composés sont appelés ***OLE containers*** et les applications qui fournissent les objets sont appelés les ***OLE servers***. Une application peut être aussi bien un ***OLE container*** et un ***OLE server***.

> *Linking and embedding* : En plus des informations statiques comme les tableaux Excel, un *OLE compound documents* peut aussi contenir des éléments actifs tels que : les données multimédia ou des services externes (exemple : les échanges boursiers). Si un objet est attaché, il réside toujours en dehors du document composé, typiquement sur un serveur où plusieurs utilisateurs accèdent à une seule version.

[1] [Ple96]

Visual Editing : Que l'on appelle aussi activation sur place. Le *visual editing* est le nom donné pour désigner le processus d'édition de l'objet serveur à l'intérieur d'un *OLE container*. Pour faire ceci il faut fusionner les menus des deux applications (*OLE container* et *OLE server*).

- ***OLE automation***

Avec *OLE automation* on a la possibilité d'avoir un programme appelant activant une procédure OLE à distance. Quand un contrôleur veut obtenir un pointeur sur un objet serveur, il le fait simplement en mettant les propriétés et les méthodes du serveur et non pas en créant un objet persistant.

- ***OLE controls***

On peut voir *OLE controls* comme des *OLE compound documents* capables d'intercepter et de gérer des événements externes. Ces événements externes sont ensuite traités par l'*OLE automation*.

- ***OLE drag-and-drop***

Disponible aussi bien dans les *OLE compound documents* et *OLE controls*, le *drag-and-drop* est maintenant une fonctionnalité essentielle dans la version Windows 95 et les versions ultérieures. Ce service OLE est de 3 types :

Inter-window : Permet de faire du *drag* d'objets d'une fenêtre d'application vers une autre.

Inter-objet : Permet de faire du *drag* d'un objet et le *drop* de cet objet vers un autre objet. Exemple transférer un tableau Excel vers un document Word.

Droping onto icons : Dans Windows 95 par exemple on peut faire du *drag-and-drop* entre les objets du bureau Windows.

4.3.4 *Distributed COM* (DCOM) [1]

COM présenté dans les paragraphes précédents supportait la communication au sein d'une seule machine. DCOM (figure 4.8[2]) gère des objets distribués au sein d'un réseau et offre des fonctionnalités équivalentes à celles offertes par l'ORB de CORBA, mais l'idée fondamentale c'est qu'il n'y a pas de *broker*. Avec DCOM on a une communication directe entre les objets à la différence de CORBA où on doit passer par l'ORB.

---

[1] [Hal96]
[2] [Ple96]

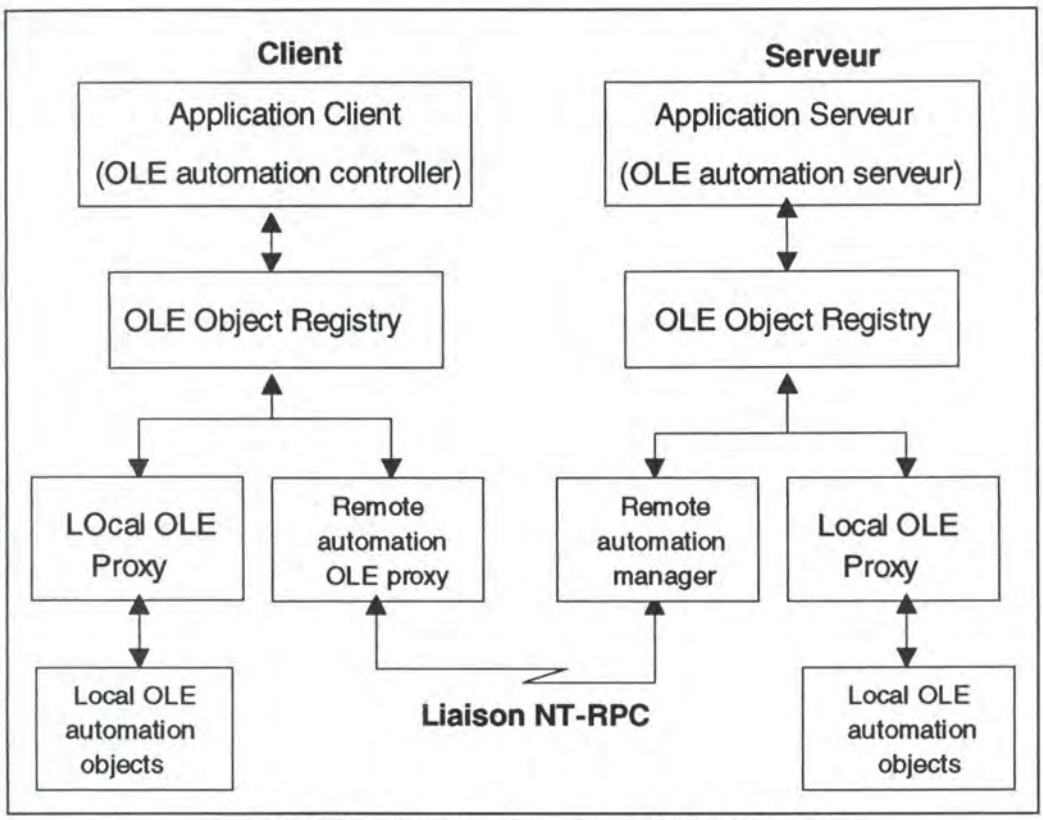


Figure 4.8 : Fonctionnement de DCOM

DCOM garde trace des composants distribués sur le réseau et permet au client d'accéder aux services offerts par ces composants.

DCOM est une couche OS qui isole les applications et les composants des détails concernant le transport à travers le réseau et le repérage des composants. Les utilisateurs n'auront pas à se soucier de l'emplacement des objets qui peuvent donc se trouver soit localement soit à distance.

Le *Remote automation* utilise un OLE *proxy* sur le client et un *Remote automation manager* sur le serveur. Quand une application cliente appelle les méthodes d'un objet OLE *automation*, elle vérifie d'abord dans l'*OLE object registry*. Le *registry* sait si l'objet appelé se trouve sur une machine locale ou sur une machine distante. Si l'objet est sur une machine locale la requête est prise en charge par le *Local* OLE *proxy*. Par contre si l'objet appelé se trouve sur une machine distante c'est le *Remote automation* OLE *proxy* qui prend en charge la requête. Si c'est le cas il traduit l'appel en un Windows NT-RPC et l'envoie à travers le réseau au serveur. C'est ensuite le *Remote automation manager* qui traduit le RPC vers un appel local à l'objet OLE. Les réponses de l'appel sont traitées de la même façon.

4.3.5 COM et CORBA

Comme CORBA, COM sépare l'interface de l'objet de son implémentation et exige que toutes les interfaces soient déclarées en utilisant un langage de définition d'interface. Mais Microsoft fournit un autre langage appelé ODL (*Object Description Language*).

A la différence de CORBA qui est basé sur le modèle objet classique, COM ne l'est pas. En effet COM ne supporte pas par exemple la notion de l'héritage multiple. Par contre un composant COM peut supporter plusieurs interfaces et permet la réutilisabilité.

Un objet COM n'est pas un objet au sens orienté objet. Les interfaces COM ne possèdent pas d'états et ne peuvent pas être instanciées pour créer un objet unique.
Une interface COM est simplement un groupement logique de fonctions sémantiquement liées. Les clients COM peuvent avoir un pointeur pour accéder aux fonctions dans l'interface, ce pointeur ne concerne pas l'état de l'information.

Comme CORBA, COM fournit des interfaces statiques et dynamiques pour l'invocation de méthodes. Le Type ***Library*** est l'équivalent dans le monde COM de l'*Interface Repository* de CORBA. Les précompilateurs COM peuvent peupler le Type *Library* par les descriptions des objets ODL définis y compris leurs interfaces et les paramètres. Les clients peuvent émettre des requêtes au Type *Library* pour découvrir quelles sont les interfaces qu'un objet supporte et quels sont les paramètres nécessaires pour invoquer une méthode particulière.

4.4 Conclusion

La technologie des objets distribués permet la communication entre objets sur le réseau. Un objet client envoie une requête à un objet serveur soit directement soit par l'intermédiaire d'un ORB qui joue le rôle de courtier entre les deux.

Cette technologie est bien appropriée pour des applications qui devront transgresser les frontières de leurs propres environnements de développement et communiqueront avec d'autres applications sur d'autres réseaux et d'autres systèmes d'exploitations. Ceci est en parfaite concordance avec le développement des réseaux à grande distance (WAN) où l'hétérogénéité est très importante.

Parmi ces environnements ouverts et hétérogènes on trouve notamment le *Web* qui est en train de passer d'un outil de distribution d'informations passives vers un outil de migration d'application. Plus encore on veut rendre le *Web* totalement client/serveur. Dans ce chapitre on a vu deux approches qui se confrontent. Dans le chapitre suivant on va voir un autre outil en l'occurrence JAVA qui permet principalement de rendre les pages *Web* (les pages HTML) totalement animées et interactives mais en combinant cet outil avec la technologie des objets distribués, on peut rendre le *Web* totalement client/serveur.

Chapitre 5 : JAVA

5.1 Introduction

Puisque le souhait est de porter le mode ***slave*** **passif** de l'application ***Whiteboard*** sur le ***Web***, parmi les outils de développement existants sur le *Web*, JAVA[1] présente un outil complet comme on va le voir à travers les caractéristiques de ce langage ainsi que l'usage que l'on peut en faire.

Dans ce chapitre on présente ce que **JAVA** peut apporter aux pages **H**yper **T**ext**M**arkup **L**anguage (**HTML**) qui sont les documents communément manipuler sur le ***Web***. Par ailleurs, vu les caractéristiques que possède le langage, plusieurs produits ont été développé sur la base de ce langage pour permettre de développer des applications client/serveur sur le *Web* soit entre un client *Web* et un serveur *Web* dans ce cas tous les deux utilisent le protocole **HTTP** soit entre un client *Web* qui utilisent le protocole HTTP et un serveur utilisant n'importe quel protocole.

Les pages actuelles que l'on visualise sur le *Web* souffrent d'un défaut majeur : ces objets passifs, codés en HTML, sont incapables d'effectuer le moindre traitement local au niveau du client *Web*. JAVA transforme le *Web* d'un simple instrument de publication passive en un réseau universel pleinement client/serveur. Le principe est d'avoir un programme compilé stocké sur le serveur HTTP, intégré dans une page HTML, cette page est téléchargée par un client *Web* à l'aide d'un outil de navigation doté d'un interpréteur Java pour exécuter le programme compilé.

La figure 5.1 présente de façon générale la communication entre un client *Web* qui est typiquement l'utilisateur qui visualise des pages HTML et un serveur *Web* qui fournit ces pages HTML.

Le client *Web* utilise les services des couches inférieures (HTTP dans notre cas). Le protocole HTTP gère la communication entre un client HTTP et un serveur HTTP on parle alors de **requête** et de **réponse**, il permet aussi de déclencher des scripts sur le serveur. HTTP utilise le protocole de bas niveau **TCP/IP**.

[1] http://java.sun.com/java.sun.com/aboutJava/index.html

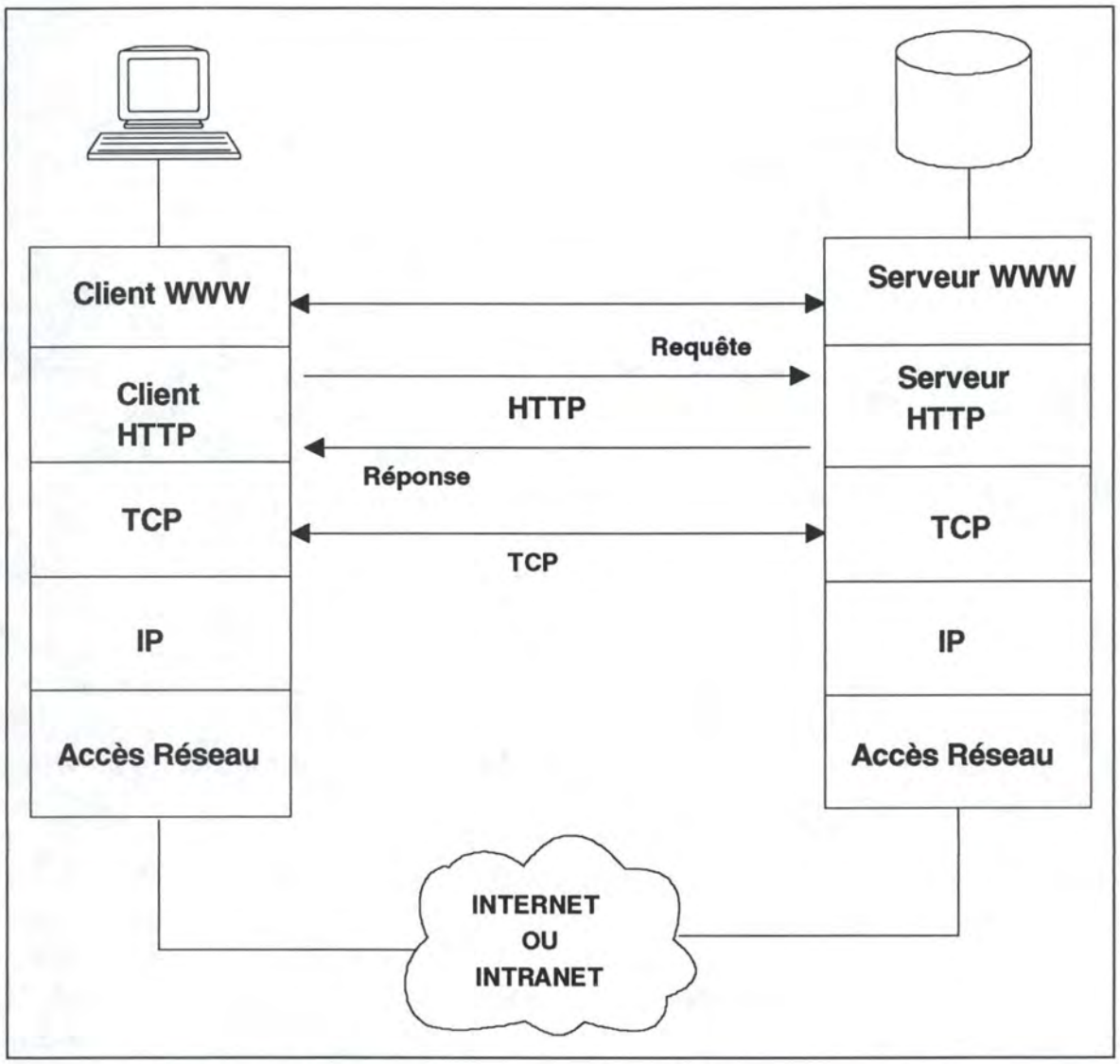


Figure 5.1 : Communication entre un client *Web* et un serveur *Web*

5.2. Les navigateurs classiques

Les navigateurs classiques tel : NCSA Mosaic fournissent une illusion d'Interactivité. En utilisant les pages HTML, ces navigateurs offrent des liens hypertextes vers d'autres pages HTML.

Le cheminement peut être résumé comme suit :

1. (*) Click sur le lien
2. Recherche de la donnée
3. Formatage et Affichage de la donnée
4. Allez (*)

Ces navigateurs sont aussi incapables de traiter n'importe quel type de donnée et n'importe quel protocole. L'utilisateur est toujours contraint d'utiliser le protocole que le navigateur supporte.

Ces protocoles sont complètement incorporés (figure 5.2) dans le navigateur et si on veut supporter un autre protocole on est obligé d'ajouter au navigateur le code supportant ce nouveau protocole.

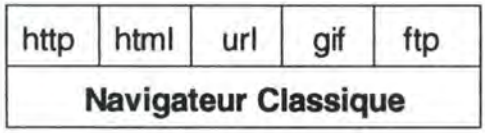


Figure 5.2 : Protocoles Supportés par les Navigateurs Classiques

On a essayé de rendre ces navigateurs classiques plus intelligents en leur permettant de lancer un programme qui est ensuite exécuté au niveau du serveur HTTP. Parmi ces tentatives il y a les scripts CGI que l'on développe par la suite.

• *Common Gateway Interface* **(CGI)**

. Définition

Le CGI permet à un client *Web* de lancer et de passer de l'information à un programme. L'utilisateur peut recevoir en retour de l'information sous forme de page HTML créée dynamiquement.

Si par exemple un utilisateur désire s'inscrire à une *mailing list* (figure 5.3), la méthode la plus simple de faire ceci sur le *Web* est d'utiliser un **formulaire** qui permet à un utilisateur de remplir des champs et de formuler des choix possibles (**phase 1**). Une fois le formulaire rempli, les informations sont passées à un programme CGI (ou Script CGI) sous forme d'une liste de paramètres (**phase 2**).

Le programme CGI exécute ce qui suit :

1. Analyser la liste des paramètres (**phase 3**).
2. Exécuter les actions nécessaires (**phase 4**).
3. Retourner l'information à l'utilisateur sous forme d'une page HTML (**phase 5**).

Le programme CGI crée dynamiquement une page HTML contenant un message confirmant que la requête a été bien exécutée. La page HTML est ensuite transmise au client *Web* (**phase 6**).

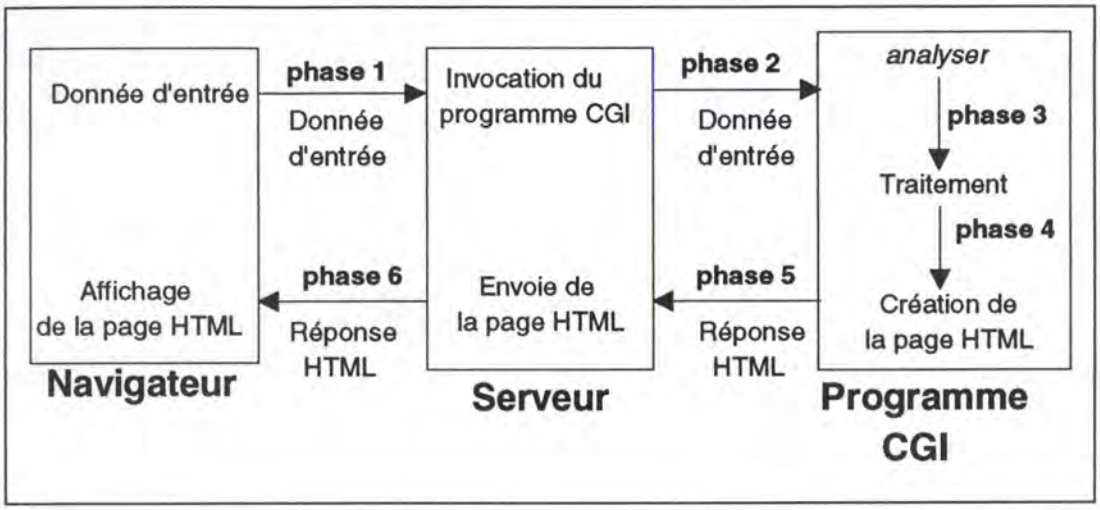


Figure 5.3 : Navigateur et CGI

Du *point de vue de l'utilisateur*, les formulaires HTML ne sont pas de véritables applications. Elles sont essentiellement textuelles et statiques et de ce fait restreignent les actions qu'un utilisateur peut accomplir. Les réponses instantanées que les utilisateurs attendent des applications ne sont pas garanties puisque les données des formulaires doivent être envoyées au serveur pour la validation. Des actions telles que : le dessin sur écran ou *drag-and-drop* ne peuvent pas être exécutées. Les mises-à-jour des écrans sont faites par rechargement de page HTML ce qui détruit le contexte utilisateur. Les utilisateurs ne peuvent pas se rendre compte facilement quand des changements au niveau d'un serveur se produisent, et donc faire des applications où la contrainte de temps est très importante avec des scripts CGI est très difficile.

Du point de vue du développeur, les formulaires et les scripts CGI sont très restrictifs. L'utilisateur ne manipule que des strings. Les types de données tels : floats, tableaux ou les objets ne sont pas utilisés.

Avec les méthodes conventionnelles les developpeurs sont incapables d'écrire des applications *Web* se comportant de la même façon que les applications traditionnelles.

5.3 Nouvelle Génération de Navigateur

Dans le reste du document on désignera par **N.C.J** (Navigateur Conforme à JAVA) un navigateurs *Web* doté d'un **interpréteur JAVA** et de **son environnement d'exécution**.

Ces navigateurs permettent non seulement une migration de l'information à travers le *Web* mais une migration d'applications écrites en langage JAVA et compilées en un format **Bytes-Code neutre**. Ces applications appelées ***Applets*** peuvent être incorporées à l'intérieur des pages HTML.

Avec les *Applets* Java, les utilisateurs *Web* peuvent construire des pages *Web* incluant animations, graphiques, jeux, et autres effets spéciaux. Ce qui permet de rendre ces pages *Web* "hautement" interactives. L'utilisateur peut en effet interagir avec une page *Web* supportant Java via la souris, le clavier et autres objets de l'interface : bouton, boîte de dialogue, champ de texte, etc...
Les pages *Web* revêtiront une autre dimension : des sites capables de faire des mises-à-jour en temps-réel au niveau d'une bourse d'échange. D'autres offrant des petites applications: tableurs ou calculatrices. Ou encore manipulant des figures géométriques.

Les *Applets* Java sont traitées par un N.C.J comme n'importe quels objets média : image, audio ou fichiers vidéo inclus dans une page **HTML**.

Caractéristiques d'un N.C.J

On appellera par la suite **host (client *Web*)** la machine qui visualise la page HTML contenant l'*Applet* et par **host d'origine (serveur *Web*)** la machine qui abrite la page HTML, le fichier **.class** de format **Bytes-Code** (qui est généré lors de la compilation d'un code JAVA) et le fichier **.html** qui référence l'*Applet* sur le serveur. Donc si quelqu'un se connecte sur le *Web* pour charger une page HTML qui contient un *Applet*, le serveur envoie le fichier .class à travers le réseau pour le *host* qui en a fait la demande. Ceci oblige le navigateur à adopter une politique de sécurité pour protéger le host sur lequel l'*Applet* s'exécute.

Un N.C.J se comporte comme une **fédération de différentes pièces** : protocoles, types de données, différentes façons de naviguer et de chercher l'information. A la différence des autres navigateurs, il sait comment se comporter avec une nouvelle pièce qui vient s'ajouter et de façon dynamique sans réécrire quoi ce soit pour supporter cette nouvelle pièce.

On peut réunir ses caractéristiques sous 3 thèmes :

- **Contenus Dynamiques**

Avec un N.C.J on peut écrire des pages HTML totalement animée. On peut visualiser des simulations et interagir avec elles en changeant des paramètres de la simulation et voir le résultat sur l'écran et ce en temps réel. On peut aussi changer les dimensions d'une figure géométriques à l'aide de la souris, faire tourner les différentes faces d'un cube.

• **Types Dynamiques** (figure 5.4)

Le N.C.J se comporte de façon dynamique avec n'importe quel nouveau type d'objet qu'un utilisateur veut visualiser sur l'écran. Si un client télécharge un objet d'un serveur en émettant une requête (**phase 1**) et si le N.C.J ne connaît pas le type (**phase 2**), il suffit que l'utilisateur ait déjà écrit un programme Java qui lui sait comment se comporter avec l'objet, le navigateur charge donc le programme Java (**phase 3**) et l'exécute avec l'interpreteur Java qui fait partie du navigateur et l'objet est affiché (**phase 4**).

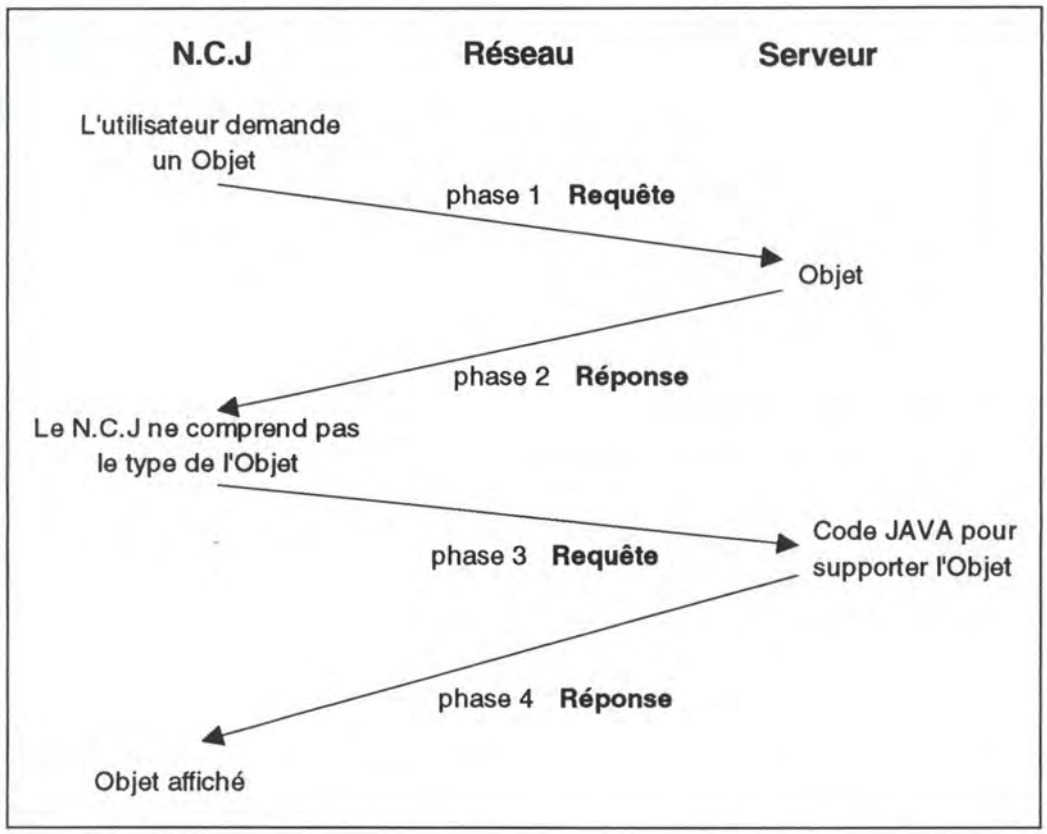


Figure 5.4 : Types Dynamiques au sein d'un NCJ

• **Protocoles Dynamiques** (figure 5.5)

Au lieu d'avoir des protocoles incorporés comme dans les navigateurs classiques, les N.C.J utilisent le nom d'un protocole et le passent au gestionnaire de protocole approprié.

Le N.C.J reçoit une référence vers une donnée via son URL. Une URL est en général de la forme **protocol://serveur/**nom_fichier, (**protocol** peut être soit HTTP ou FTP ou un **nouveau protocole** qu'un vendeur veut utiliser par exemple pour chiffrer des données sensibles pour raison de sécurité). Le N.C.J cherche le gestionnaire de ce **protocol** d'abord dans le système de fichier et s'il ne le trouve pas, il cherche dans le **serveur**.

Ce gestionnaire est un programme Java qui se charge d'envoyer les données, ceci est tout à fait transparent à l'utilisateur qui peut donc spécifier au navigateur des URL sans se préoccuper si le navigateur supporte tel ou l'autre protocole. C'est au vendeur par exemple dans le cas d'un **nouveau protocole** de fournir le **gestionnaire de son protocole**, le N.C.J se charge de l'exécuter puisque l'interpreteur Java est incorporé dans le N.C.J.

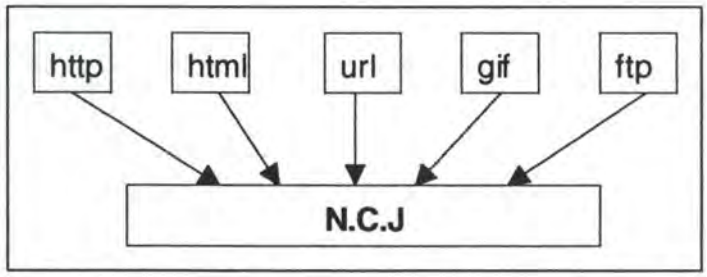


Figure 5.5 : Protocoles Dynamiques

5.4 Caractéristiques du langage JAVA

Le langage Java permet de développer des applications appelées ***Applet*** qui s'exécutent dans le contexte d'une page HTML chargée par n'importe quel N.C.J. On peut aussi développer des programmes ***stand alone*** qui s'exécutent par l'interpreteur à partir d'une ligne de commande sans passer par le *Web*.

Sun dispense le compilateur **javac** pour générer un fichier en format Bytes-Code qui ne dépend pas d'une machine particulière, un **interpréteur**, un **debuggeur** et un outil de visualisation : ***Appletviewer*** pour tester ces *Applets*.

On va étudier le langage à travers ses caractéristiques :

• **Orienté-Objet**

Le besoin de développer des systèmes distribués exige l'utilisation de différents paradigmes de la programmation Orientée Objet, tels : l'**héritage**, le **polymorphisme**, l'**encapsulation**, la **communication entre objets** par invocation de méthode, la notion d'**Interface**.

Ces concepts généraux de la programmation orientée objet se retrouvent dans le langage JAVA. Mais JAVA ne supporte pas directement l'héritage multiple.
Sauf les types de données simples (boolean, byte, char, short, int, long, float, double), tout est un objet, y compris les tableaux et les Strings.

• **Distribué**

Java possède des classes pour supporter différentes applications reposant sur le protocole TCP/IP exemple : HTTP, FTP. Sous Java une application accède à un objet à travers le réseau via son URL avec la même facilité qu'une application accédant à un fichier local.

• **Architecture neutre, Portabilité et Robustesse**

Avec le développement actuel des réseaux et notamment Internet, le souci majeur des developpeurs est de construire des applications distribuées. Ces applications seront amenées à migrer à travers différents types de systèmes informatiques, différents types de processeurs et différents types de systèmes d'exploitation. Ces applications doivent aussi faire face à une variété d'interface utilisateur. Pour résoudre ces différents problèmes il est nécessaire que les applications soient neutres vis-à-vis d'une architecture particulière, et qu'elles soient portables.

. **Architecture neutre :** Le compilateur Java génère un code compilé de format Bytes-Code. Ce code est indépendant d'un langage machine particulier. Le but est d'avoir une seule version d'une même application qui puisse s'exécuter aussi bien sur IBM PC, Apple Macintosh que sur une station UNIX.

. **Portabilité :** Le premier avantage d'utiliser le Bytes-Code c'est la portabilité du code compilé à n'importe quel système qui possède l'interpreteur Java et l'environnement d'exécution. Le fait que le langage soit neutre par rapport à une architecture particulière ne garanti pas une portabilité totale. Reste le codage des types de données simples. Java utilise les formats communs à tous les processeurs qui se trouvent sur le marché.

Ainsi les types de données simples sous Java sont codés comme suit :

byte 8-bits two's complément
short 16-bits two's complément
int 32-bit two's complément
long 64-bit two' s complément
float 32-bit IEEE 754 floating point
double 64-bit IEEE 754 floating point
char 16-bit Caractère Unicode.

Un autre aspect qui permet la portabilité du langage, ce sont les librairies qui font partie de l'environnement d'exécution du langage Java. Par exemple la classe **System** permet une indépendance vis-à-vis des ressources d'une machine particulière. Les librairies **AWT** (**A**bstract **W**indow **T**oolkit) permettent la spécification d'outils abstraits de l'interface.

L'environnement Java est aussi totalement portable.En effet le compilateur est lui-même écrit en Java. L'environnement d'exécution est écrit en ANSI C.

. **Robustesse** : Les *Applets* Java sont amenés à "surfer" à travers le *Web*, à s'exécuter sur des *hosts* qui ne connaissent pas forcément comment l'*Applet* va se comporter avec les ressource locales. JAVA étant un langage fortement typé, il permet une vérification efficace lors de la phase de compilation ce qui permet de nettoyer le code bien avant son interprétation. A la différence de C/C++, JAVA élimine le risque d'écraser des données en mémoire ou de manipuler des données corrompues. Et on ne peut pas faire du *cast* d'un entier vers un pointeur.

• **Langage Performant**
Le code compilé doit passer par deux étapes préalables avant d'être interprété : la **vérification du Bytes-Code** (5.7.2) et le **Bytes-Code *loader*** (5.7.3) lors du chargement d'une classe référencée.

Ceci assure une grande performance lors de l'interprétation. En effet l'interpréteur agit avec une grande vitesse sans vérification de l'environnement d'exécution.

Pour la gestion de la mémoire, l'environnement JAVA utilise une ***garbage collection*** qui exécute un **thread** de faible priorité en ***background***. Ce thread rassemble et compacte l'espace mémoire inutilisé, c'est donc la *garbage collection* et non l'utilisateur qui gère l'espace mémoire en déterminant les espaces mémoire qui n'ont pas été référencés pendant une certaine période.

• **Interprété**
Java est un langage interprété et donc neutre par rapport à une architecture particulière. En effet le compilateur Java génère un fichier de format **Bytes-Code** (ne contenant aucun détail d'implémentation) pour la Machine Virtuelle Java.

• **Notion d'Interface**
Le langage Java implémente le concept d'interface emprunté à Objective C. Une interface est une collection de méthodes qu'une classe doit implémenter. Une interface ne définit pas d'instances de variables ou d'implémentation. On peut utiliser de multiples interfaces sans se soucier des difficultés créées par une structure d'héritage standard. Un objet JAVA peut implémenter plusieurs interfaces et on peut donc faire indirectement de l'héritage multiple.

5.5 *Applet* : Migration d'Applications sur le *Web*

5.5.1 Fonctionnement d'un *Applet*

Un *Applet* est un programme Java qui s'exécute dans le contexte d'une page HTML. Le compilateur génère un fichier **.class** de format Bytes-Code à partir du code écrit en JAVA. Ce fichier Bytes-Code est incorporé dans une page HTML via l'étiquette ***<Applet>***. Cette étiquette est utilisée pour décrire l'*Applet*, sa **longueur**, sa **largeur** et ses **paramètres**. Quand le navigateur rencontre l'étiquette *<Applet>*, il télécharge le code compilé de l'*Applet* et l'exécute.

Exemple :

<title> BAZZI Home Page </title>
<h1> Bienvenue </h1>
***<Applet* code=Slave.class width=800 height=800>**
</Applet>
<address>mba@info.fundp.ac.be</address>

Dans cette page HTML on constate la présence d'une étiquette *<Applet ...></Applet>* ceci permet à une page HTML d'incorporer du code compilé en langage neutre (Bytes-Code).

Un navigateur doté d'un interpreteur JAVA peut donc exécuter ce code. Dans l'exemple le code compilé est **Slave.class**.

Un *Applet* comme n'importe quel autre objet média prend un certain temps avant d'être chargé. Et une fois qu'il est chargé, il s'exécute comme n'importe quelle autre application sur le *host*. L'*Applet* peut aussi télécharger d'autres ressources (images, fichiers audio ou d'autres classes Java). Il peut aussi interagir avec le *host*.

5.5.2 L'étiquette html : *<Applet >*

L'étiquette *<Applet>* est proche de l'étiquette ***<img>***. Comme *<img>*, *<Applet>* référence un fichier qui ne fait pas partie de la page HTML dans laquelle elle est incorporée.
Pour *<img>*, le paramètre **src** permet de retrouver le fichier image. Pour *<Applet>* le paramètre **code** joue le même rôle. Le paramètre **code** renseigne le navigateur sur l'emplacement où il faut chercher le code compilé c'est à dire le fichier **.class**. Donc si on charge la page HTML avec l'URL **http://www.info.fundp.ac.be/~mba/slave.html** et si code est Slave.class alors le navigateur va chercher le fichier **.class** dans l'URL : **http://www.info.fundp.ac.be/~mba/Slave.class**

Par contre si l'*Applet* réside à un endroit différent de celui de la page HTML **slave.html** alors un autre paramètre intervient c'est **codebase**. Le codebase spécifie le **répertoire** qui contient le fichier **.class** alors que **code** spécifie le **nom** du fichier **.class**.

De même que pour **img**, on peut positionner l'*Applet* dans la page en utilisant les paramètres suivants : **height**, **width**, **align,** etc...

5.5.3 La Communication entre un *Applet* et le N.C.J

La partie entre ***<Applet>*** et ***</Applet>*** peut être utilisée pour permettre à un N.C.J de passer des paramètres à des *Applets*. Ceci se fait par le biais de l'étiquette **param**. A l'intérieur de l'*Applet* ce paramètre est passé en invoquant la méthode **getParameter** de la class java.*applet.Applet*.

Le passage de paramètre est très utile si on veut que l'*Applet* donne d'autres résultats sans pour autant recompiler l'*Applet*.
Il n'y a pas de limite sur le nombre de paramètres qu'on peut passer à une *Applet*.
Tous les paramètres sont passés comme des strings. Si on veut passer un autre type, il faut le passer d'abord comme string et après le convertir au type désiré.

5.5.4 Cycle de vie d'un *Applet* [1]

Le cycle de vie d'un *Applet* est constitué par l'ensemble des événements que l'*Applet* reçoit à partir du moment où il est chargé jusqu'au moment où il n'est plus accessible et donc détruit par le système. La figure 5.6 présente le cycle de vie d'un *Applet*.

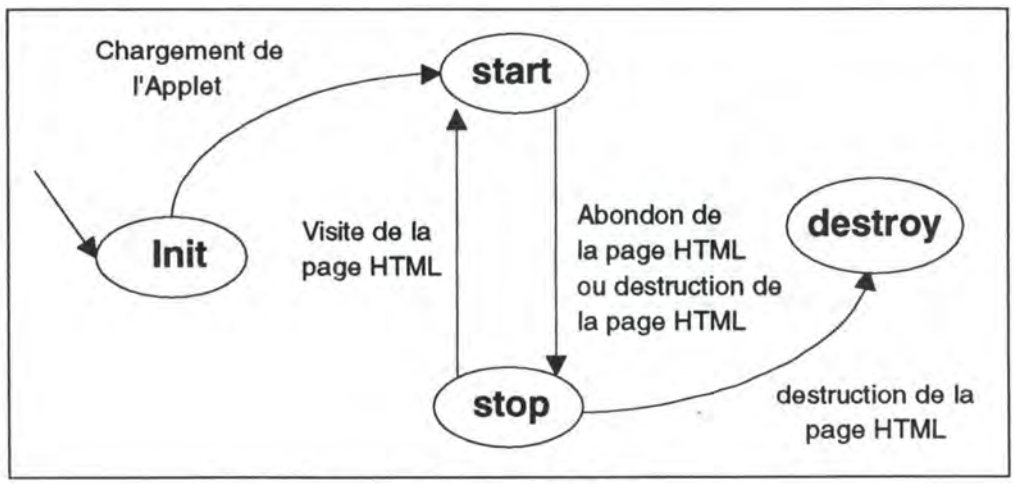


Figure 5.6 : Cycle de vie d'un *Applet*

[1] [Van96] p : 122-123

Les méthodes (ou événements) sont :

init()
La méthode **init()** est appelée quand l'*Applet* commence son exécution. Le N.C.J utilise cette méthode en cas de rechargement de l'*Applet* ou quand on retourne à la page HTML après l'avoir quitté. Elle est appelée automatiquement par le système.
La méthode **init** est généralement réécrite en utilisant le mécanisme du polymorphisme, on peut par exemple recevoir des paramètres de la page HTML en invoquant la méthode
getParameter(String str) dans le corps de la méthode **init**.

start()
La méthode **start()** est automatiquement appelée après que la méthode **init** aura terminé son travail. Elle est appelée automatiquement par le système.
De même que pour la méthode **init**, la méthode **start** est généralement réécrite, on peut par exemple initialiser des variables dans le corps de la méthode **start**.

stop()
La méthode **stop()** est appelée lorsque l'utilisateur quitte la page HTML.
Elle est appelée automatiquement par le système.

destroy()
Cette méthode est appelée juste après la méthode stop() pour faire le dernier "ménage" avant que l'*Applet* ne soit définitivement quittée.

5.6. La machine virtuelle JAVA

Le but de la machine virtuelle JAVA est de permettre la portabilité des applications JAVA. Grâce à la machine virtuelle JAVA les applications JAVA peuvent s'exécuter sans se soucier ni du type de système d'exploitation ni du type de processeur.

Après la compilation d'un programme JAVA, on obtient un fichier de format Bytes-Code. Ce fichier obtenu ne contient aucun détail d'implémentation et de ce fait il ne dépend d'aucun système d'exploitation, ni d'aucun processeur particulier.

La machine virtuelle JAVA prend ce code compilé en Bytes-Code et l'exécute. La couche de bas niveau de la machine virtuelle JAVA traduit ce pseudo-code machine en code machine réel de l'environnement physique sur lequel la machine virtuelle s'exécute, c'est donc la machine virtuelle JAVA qui gère le dialogue avec les ressources physiques du système y compris le système d'exploitation.

Dans la figure 5.7 on présente le cas de l'exécution d'un programme classique. Dans ce cas le programmeur spécifie directement les ressources physiques en exécutant des appels systèmes au système d'exploitation, le programmeur utilise les pointeurs pour accéder aux données en mémoire. Ceci peut parfois engendrer des pertes de données en cas de mauvaise manipulation des pointeurs.

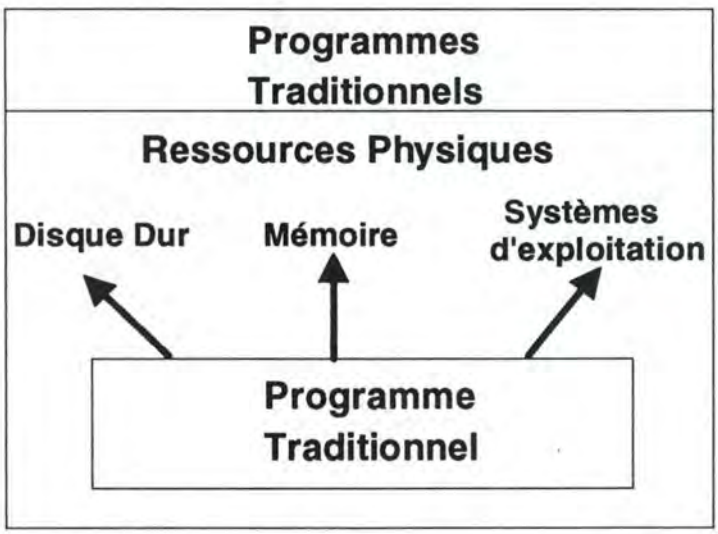


Figure 5.7 : Programme traditionnel et ressources Système

Par contre dans la figure 5.8 le programmeur spécifie des ressources abstraites et c'est la machine virtuelle JAVA qui se charge de traduire les demandes du programme JAVA en ressources physiques de l'environnement réel d'exécution.

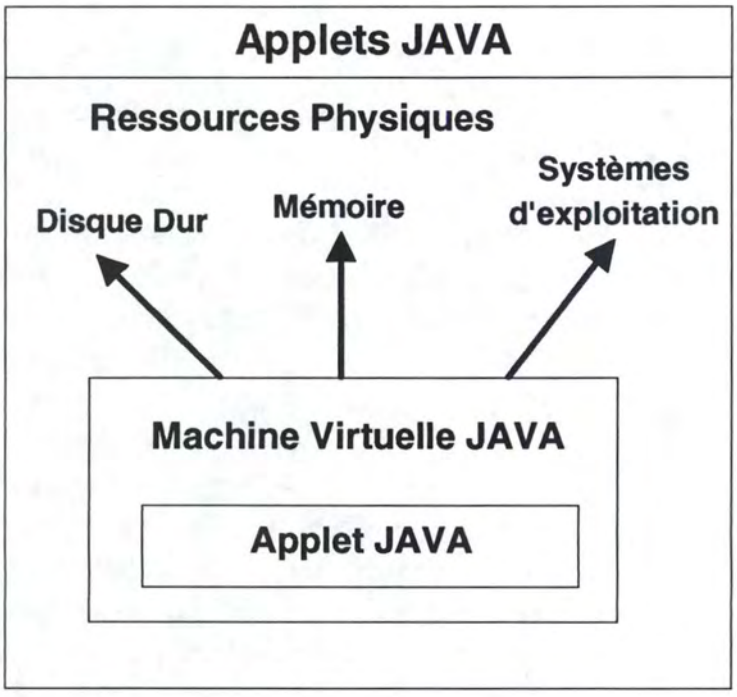


Figure 5.8 : Machine Virtuelle JAVA

5.7 Sécurité dans l'environnement Java

La portabilité du langage permet de charger dynamiquement des applications Java à travers le réseau et de les exécuter localement. En particulier, des *Applets* peuvent être chargés et exécutés par un utilisateur en train de "surfer" sur le *Web* via un N.C.J. Mais se pose un gros problème celui de la sécurité du host.

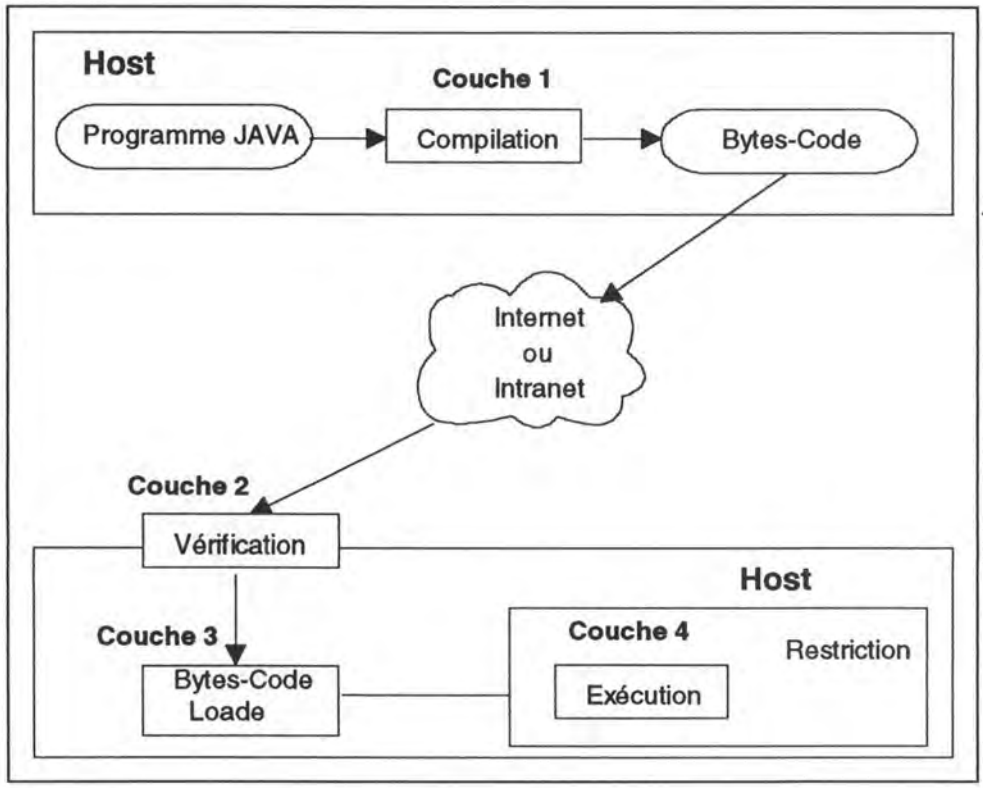


Figure 5.9 : Les couches de sécurité dans l'environnement JAVA

Un code java doit passer par 4 couches de sécurité, à savoir le langage et le Compilateur, le Vérificateur du Bytes-Code, la Sécurité au niveau du Bytes-Code Loader et la Sécurité propre aux navigateurs (figure 5.9) :

5.7.1 Couche 1 : Le langage et le Compilateur

La première ligne de défense du compilateur est l'allocation d'espace mémoire et le mode de référence. La taille mémoire n'est pas décidée lors de la compilation comme en C++ mais au *run-time* et ceci dépend de la plate-forme sur laquelle on exécute le programme Java et notamment le système d'exploitation.

Java ne manipule pas les pointeurs comme en C/C++ c'est à dire des cellules mémoires qui contiennent les adresses d'autres cellules mémoire. Un code ne peut pas corrompre des espaces mémoire puisque le modèle de référence est totalement opaque au programmeur et contrôlé entièrement par le système sous-jacent lors de l'exécution.

5.7.2 Couche 2 : Vérificateur du Bytes-Code

Avant d'exécuter le Bytes-Code, il faut s'assurer que c'est bien un compilateur Java **Agréé** qui l'a produit. Si le Bytes-Code a été produit par une autre source que le compilateur Java, il faut s'assurer qu'il ne viole pas les normes de sécurité imposées par Java.
L'environnement d'exécution fait donc passer le Bytes-Code par le **vérificateur du Bytes-Code.**

Le vérificateur du Bytes-Code exécute ce qui suit :
1. Vérifie le format du fichier .class.
2. S'assure que l'on n'essaye pas de contrefaire l'utilisation de pointeurs (sous JAVA on ne manipule pas de pointeurs).
3. S'assure qu'on ne viole pas les accès aux objets et que l'encapsulation des données est bien respectée.

Après ce test on est sûr que le code compilé vérifie les propriétés suivantes :
. Il n'y a pas de stack overflows d'une opérande ou débordement de capacité
. Les types des paramètres de toutes les instructions sont connus et sont corrects.
. Les différents types d'accès aux objets sont légaux (privé, public et protégé).

Maintenant l'interpreteur peut opérer plus rapidement puisqu'il n'y a plus rien à vérifier.

5.7.3 Couche 3 : Sécurité au niveau du Bytes-Code Loader

Lors de l'exécution du Bytes-Code, celui-ci peut faire des références à des classes qu'il faudra donc charger, soit du système de fichier local soit du réseau. Après que le code chargé passe le test de vérificateur du Bytes-Code, la prochaine ligne de défense sera le **Bytes-Code *Loader.***

On peut voir l'environnement d'exécution comme des suites de classes séparées par des ***name spaces***. Toutes les classes qui proviennent du système de fichier local sont rassemblées dans un seul ***name space*** et de ce fait elles peuvent définir leur propre politique de sécurité. Alors que les classes en provenance de *hosts* différents appartiennent à des ***name space*** différents et ce fait elles sont assujettis aux restrictions définis par le navigateur qui charge l'*Applet*. Quand une classe est chargée à partir du réseau elle est placée dans le ***name space*** privé associé à son origine.

Quand une classe fait référence à une autre classe, elle consulte d'abord le *name space* local, puis celui de la classe référencée.

5.7.4 Couche 4 : Sécurité propre aux navigateurs

Cette couche est implémentée par le navigateur lui-même. Donc cette phase concerne les *Applets* JAVA c'est à dire les programmes destinés à être exécutés dans le contexte des N.C.Js.

Un N.C.J doit garantir la sécurité des ressources qui résident dans le système sur lequel l'*Applet* va s'exécuter. Le *host* doit donc pouvoir visualiser des *Applets* Java sans que les ressources du système soient menacées par le code importé.

Mais quelles sont les ressources du host qu'il faut protéger ?

On peut énumérer les ressources suivantes :

Système de fichier : l'accès au système de fichier doit être bien protégé. On peut, sous l'outil ***appletviewer*** fourni par les laboratoires Sun, rendre cette restriction plus flexible en spécifiant les fichiers auxquels on désire enlever cette contrainte.

Réseau : la création de sockets doit être bien gérée. Sous les N.C.Js on peut ouvrir un socket seulement vers le ***host*** **d'origine**.

Espace Mémoire : La protection de la mémoire est assurée par le langage lui-même (couche 1 figure 5.9).

Device de sortie : Un *Applet* ne peut pas utiliser directement les devices de sortie, il doit obligatoirement passer par les librairies Java qui gèrent ces devices.

Device **d'entrée :** Pour le moment seulement la souris et le clavier sont gérés lorsque l'événement se passe à l'intérieur de la fenêtre abritant l'*Applet*. Les *devices* comme les microphones et les caméras vidéo ne sont pas encore gérés.

Contrôle des threads : Un thread n'est pas autorisé à joindre un threadgroup auquel il n'appartient pas. La priorité d'un thread ne peut pas dépasser 10.

Environnement Utilisateur : L'accès aux variables d'environnement est bien protégé pour préserver l'intégrité de l'environnement du host sur lequel l'*Applet* exécute.

Appel Système : Les appels systèmes doivent être vérifiés. Des appels système tel ***exit*** peuvent causer la fin du processus qui lance le navigateur.

Il y a une classe Java **SecurityManager** qui contient des méthodes pour contrôler les accès à ces différentes ressources. Le tableau 5.1 regroupe les différentes méthodes qui garantissent l'intégrité des ressources du host.

Méthode	**Description**
getInCheck	détermine si la vérification est en cours
checkCreateClassLoader	empêche l'installation de plusieurs ClassLoaders
checkAccess	vérifie si un thread ou un thread group peut modifier de thread group
checkExit	vérifie si la commande Exit peut être exécutée
checkExec	vérifie si la commande système peut être exécutée
checkLink	vérifie si une librairie dynamique peut être attachée (utilisé pour native code)
checkRead	vérifie si on peut lire un fichier
checkWrite	vérifie si on peut écrire dans un fichier
checkConnect	vérifie si une connexion réseau peut être créée
checkListen	vérifie si un certain port peut être écouté
checkAccept	vérifie si une connexion réseau peut être acceptée
checkProperties	vérifie si des propriétés système peuvent être lues
checkTopLevelWindow	vérifie si une fenêtre doit avoir un *warning* spécial
checkPackageAccess	vérifie si on peut accéder à un package donné
checkPackageDefinition	vérifie si une nouvelle classe peut être ajoutée

Tableau 5.1

5.8 Mécanisme des Threads[1]

Une autre caractéristique du langage JAVA est l'utilisation des threads. Les threads permettent le lancement de différentes tâches au sein d'un même programme sans se soucier du mécanisme de gestion des processus sous-jacent. On peut utiliser soit des threads indépendants dans ce cas aucun mécanisme de contrôle d'accès n'est nécessaire, soit le concept du *multithreding* c'est à dire plusieurs threads qui doivent se synchroniser lors de l'exécution.

5.8.1 Définition

Un **thread** appelé aussi processus allégé possède un point d'entrée, un enchaînement séquentiel d'instructions et un point de sortie mais à la différence d'un programme ordinaire, un thread ne s'exécute pas de sa propre initiative il doit être exécuté à l'intérieur d'un programme.

Mais ce qui est plus intéressant c'est d'exécuter plusieurs threads en "même temps", de façon concurrente et accomplissant chacun une tâche particulière au sein d'un même programme.

Les N.C.Js sont des exemples particuliers d'applications à plusieurs threads. On peut défiler une page HTML au même moment qu'elle charge un *Applet* ou une image; entendre des données audio et voir de l'animation à l'écran, imprimer une page en background tout en chargeant une autre page; regarder 3 algorithmes de tri animés à l'écran.

5.8.2 Attributs d'un thread

Le langage Java offre une classe **Thread** qui implémente les threads JAVA de façon indépendante par rapport à un système particulier. On peut définir un thread suivant les concepts suivants : le **corps du thread**, l'**état du thread**, la **priorité du thread** et le **thread daemon**.

Corps d'un thread

Toutes les actions qu'un thread exécute doivent être déclarées dans la méthode **run()**. Après la création d'un thread et son initialisation, l'environnement d'exécution appelle la méthode **run()**. Les instructions dans la méthode **run()** reflètent le comportement du thread.
Il y a deux façons d'utiliser la méthode **run()** :

1. **Hériter** de la classe **Thread** définie dans le package **java.lang** et réécrire la méthode run() en utilisant le mécanisme du polymorphisme.

[1] http://java.sun.com/books/Series/Tutorial/java/threads/index.html

2. **Implémenter l'interface Runnable** qui se trouve dans le package **java.lang**. Cette façon de faire est très utile lorsque le programme doit hériter de plusieurs classes.

JAVA ne supporte pas l'héritage multiple, donc une classe peut hériter de la classe **Applet** et implémenter l'interface Runnable pour utiliser le mécanisme des threads.

État d'un Thread

Un thread peut se trouver dans l'un des états suivants : **Crée**, **Exécutable**, **Non Exécutable** ou **Stoppé**.

La figure 5.10 illustre les états par lesquels passe un thread et les différentes transitions entre ces différents états.

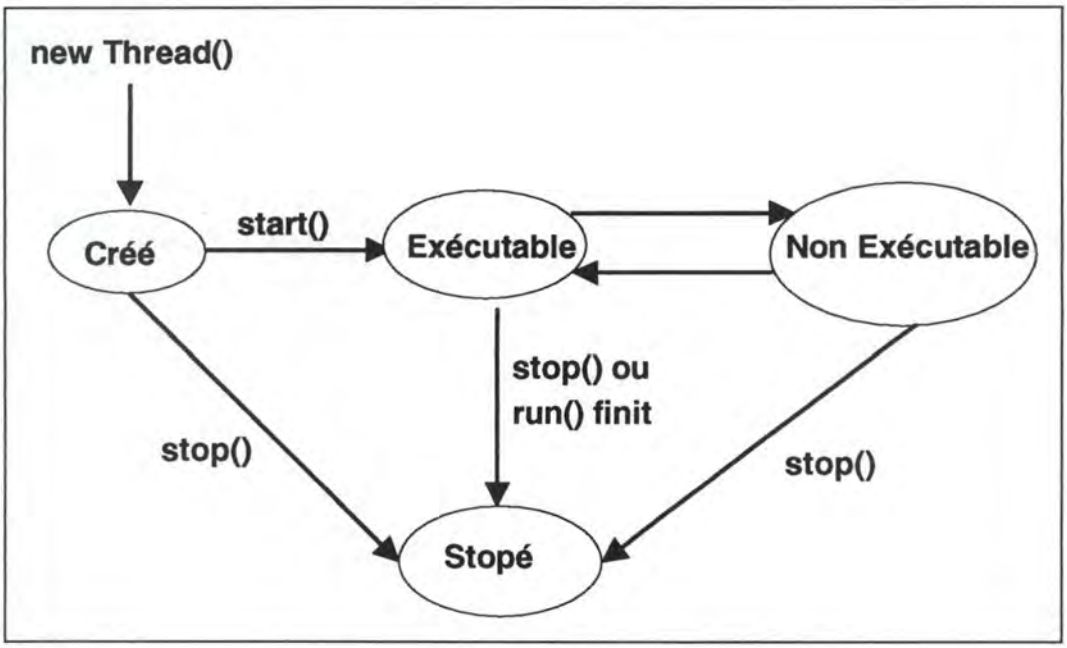


Figure 5.10 : Cycle de vie d'un thread

• **État Crée :**
L'instruction **Thread ic = new Thread(this,"La classe Image")** par exemple crée un nouveau thread. Le thread se trouvant dans cet état ne possède pas de ressources système pour s'exécuter. A partir de cet état on peut soit appeler la méthode **start()**, soit la méthode **stop()**.

• **État Exécutable :**
Soit les instructions suivantes :

```
Thread ic = new Thread(this,"La classe Image");
ic.start();
```

La méthode **start()** crée les ressources système nécessaires pour exécuter le thread, met le thread dans la file d'attente et appelle la méthode **run()**. Cet état est intitulé **Exécutable** et **non Exécuté** car le thread ne sera pas forcément élu par le *scheduler* pour détenir le CPU.

Dans les configurations actuelles il n'y a qu'un seul CPU et donc qu'un seul thread qui s'exécute à un moment donné. Donc on aura plusieurs threads dans l'état **Exécutable** mais un seul thread qui s'exécute réellement.

• **État Non Exécutable :**
Un thread entre dans cet état suite aux événements suivants:
1. La méthode **suspend()** a été invoquée.
2. La méthode **sleep()** a été invoquée.
3. Le thread a invoqué la méthode **wait()** pour qu'un autre thread puisse changer la valeur d'une variable de condition.
4. Le thread est bloquée sur une **opération I/O**

Exemple :

```
Thread ic = new Thread(this,"La classe Image")
ic.start();
try {
        ic.sleep(10000);
        }catch (InterruptedException e) {
}
```

L'instruction en gras met le thread ic en état de sommeil pendant 10 secondes. Pendant ces 10 secondes, même si le processeur devient disponible le thread **ic** ne sera pas élu.

Un thread quitte l'état **Non Exécutable** dans l'une des conditions suivantes :

1. Si un thread a été suspendu, on doit invoquer la méthode **resume()** pour permettre au thread de revenir à l'état **Exécutable**.

2. Si le thread a été placé en état de sommeil, le nombre de seconde spécifié doit s'écouler avant que le thread ne puisse revenir à l'état **Exécutable**.

3. Si le thread attend (par la méthode **wait()**) le changement de la valeur d'une variable de condition, le thread qui détient la variable doit relâcher celle-ci en invoquant la méthode **notify()** ou **notifyAll()**.

4. Si le thread est bloquée sur une I/O, l'opération doit finir.

• **État Stoppée :**
Un thread est stoppé pour deux raisons :

1. La méthode run() termine l'exécution des instructions.

Exemple :
```
public void run(){
        int i = 0;
        while(i < 100) {
                i++;
                Sytem.out.println("i = " + i);
        }
}
```

Lorsque la boucle while est exécutée 100 fois le thread qui implémente la méthode run() est mis dans l'état **stoppé**.

2. On a invoqué de la méthode **stop()**.

Exemple :
```
Thread ic = new ImageClass(this,"La classe Image")
ic.start();
try {
        ic.sleep(10000);
 }catch (InterruptedException e) {
}
ic.stop();
```

Après l'expiration des 10 secondes, la méthode **stop()** invoquée a pour effet de lancer la classe **ThreadDeath** pour tuer le thread ic. Ceci se fait de manière **asynchrone**. Le thread sera stoppé quand il recevra réellement l'exception de la classe **ThreadDeath**.

Priorité des threads

Lorsqu'un thread est crée, on lui attribue la priorité du thread qui l'a crée. On peut à n'importe quel moment changer la priorité d'un thread en invoquant la méthode **setPriority()**.
Un thread peut avoir comme priorité minimale la valeur **MIN_PRIORITY**, comme priorité moyenne la valeur **MOY_PRIORITY** ou comme priorité maximale la valeur **MAX_PRIORITY**.

L'environnement d'exécution choisi parmi les threads dans l'état **Exécutable** celui qui a la plus grande priorité. Seulement lorsque le thread passe dans l'état **Non Exécutable** ou quand il est stoppé qu'un autre thread de priorité plus inférieure est exécuté.

Le *scheduler* est aussi préemptif. Si à n'importe quel moment un thread de priorité supérieure entre dans l'état Exécutable, il est élu et préempte les autres threads.

Règle : **A n'importe quel moment, le thread de priorité supérieure par rapport aux autres threads Exécutables est élu.**

Threads Daemon

Un thread peut être déclaré comme ***daemon***. Les threads daemons sont des fournisseurs de services aux autres threads ou aux autres objets qui s'exécutent dans le même programme que le thread *daemon*. Par exemple le N.C.J implémente un thread daemon appelé : Lecteur d'Image en Background (***Background Image Reader***).
Ce *daemon* lit les images d'un système de fichier ou du réseau pour le compte d'un thread ou tout autre objet. Ce qui permet à un utilisateur de défiler une page en même que le navigateur est en train de charger l'image.
La méthode **setDaemon()** déclare un thread comme *daemon*. La méthode **isDaemon()** permet de tester si un thread est *daemon* ou non.

5.8.3 Concept du *multithreading*

Jusqu'à présent on a défini le concept de thread et de plusieurs threads qui s'exécutent de façon indépendante l'un par rapport à l'autre. Dans les environnements actuels, on sort de cette conception traditionnelle dans la gestion des événements : faire une seule chose à la fois, au profit d'un environnement concurrentiel où plusieurs événements surgissent en même temps.

L'utilisation de plusieurs threads partageant la même structure de donnée et s'exécutant de manière synchrone s'impose dans plusieurs applications. C'est ce qu'on appelle du ***multithreading***. Le cas typique c'est l'accès concurrent à un fichier. Un thread qui écrit dans le fichier et l'autre qui lit du même fichier. JAVA supporte le multithreading au niveau du langage lui-même, de son environnement d'exécution et au niveau de l'objet thread.

Au niveau du langage, les méthodes ou les blocs de programmes déclarés avec le mot réservé ***synchronized*** ne s'exécutent pas de façon concurrente. Les blocs et les méthodes précédés par le mot réservé ***synchronized*** sont de ce fait des **sections critiques**. JAVA utilise le concept de **moniteur** et de **variable de condition** introduit par C.A.R. Hoare pour régler les accès concurrents. Quand une méthode déclarée ***synchronized*** s'exécute, elle acquiert le moniteur empêchant de ce fait l'autre méthode de s'exécuter.

5.9 Le *Web* et le Client/Serveur

Dans les paragraphes présentées jusqu'à présent, on s'est focalisé sur les caractéristiques et les composants du langage JAVA et de son environnement d'exécution. On a vu aussi comment ce langage peut être à la base d'une infrastructure *Web* capable de migrer non seulement des informations passives mais aussi des applications compilées en un langage neutre, ces applications sont capables de s'installer dans l'environnement d'exécution du client *Web*, de s'exécuter et de terminer leurs exécutions. Mais ces applications appelées *Applets* ne garantissent pas une communication permanente avec le serveur HTTP et donc faire du client/serveur sur le Web en utilisant seulement JAVA et le protocole HTTP ne garantit pas les performances des applications client/serveur traditionnelles.

Néanmoins en combinant JAVA avec d'autres technologies on peut offrir aux applications *Web* des outils nécessaires pour faire du vrai client/serveur.

5.9.1 Les conditions pour faire du client/serveur sur le *Web*

On peut énumérer les 4 conditions suivantes pour faire du client/serveur sur le *Web*[1] :

1. interactivité. Le client et le serveur d'une application client/serveur sur le *Web* doivent être capables d'interagir. Cette interaction doit être garantie sans avoir à reconstruire des pages HTML ou à rétablir la connexion. Il faut rappeler que le protocole HTTP est **sans état (*stateless*)** c'est à dire qu'une fois le serveur envoie une réponse au client la connexion transport est coupée et donc le serveur HTTP ne garde aucune trace de l'échange précédent. Il n'est pas pratique d'avoir à recharger chaque fois une page HTML dans une application temps-réel telle suivre le cours dans une bourse d'échange.

2. Connexions aux serveurs. Une communication directe entre les clients *Web* et les serveurs est exigée. Le serveur HTTP ne supporte pas le concept de session ou de transaction entre le client et le serveur ce qui limite l'interactivité des applications et le genre d'application à gérer puisque la plupart des applications telles que la facturation et la gestion de stock nécessitent l'utilisation du mécanisme des transactions.

3. Utilisation du Web pour développer des applications client/serveur. Utiliser le *Web* pour le développement des applications client/serveur n'implique pas la création d'autres modèles client/serveur spécifique pour le *Web*. On doit être capable de supporter les systèmes déjà existants. Donc un langage neutre pour définir les interfaces entre les nouveaux et les anciens logiciels doit être fourni.

4. *Zero Administration* pour les clients. Le *Web* permet de localiser les clients n'importe où sur le *Web*. Ce modèle exige que les plates-formes des clients soient facilement configurables pour exécuter de nouvelles applications sur des sites distants. Une *Zéro Administration* ne nécessite aucune installation préalable pour exécuter une application.

---

[1] http://www.sun.com/sunsoft/neo/external/whitepapers/Joe-wp-new.html

5.9.2 Construction d'un système client/serveur *Web*

Construire un système client/serveur sur le *Web* nécessite les conditions énumérées en haut. Les étapes pour créer un système client/serveur opérationnel sont :

1. La création d'un modèle intégré de tous les environnements client/serveur. Ceci permet d'intégrer les nouvelles applications *Web* avec les environnements client/serveur existants.

Ceci est d'une utilité extrême sinon les developpeurs auront à réécrire des fonctionnalités qui existaient déjà. Il est donc nécessaire de créer un modèle unifié. La meilleure façon de faire ceci est d'utiliser un modèle **client/serveur à trois niveaux** (***three-tier client/server modeling***) pour créer des systèmes permettant une séparation explicite de la **partie présentation (interface)**, de la **logique de l'application** et de la **gestion des données**. Avec ce modèle en place une interface commune aux deux derniers niveaux (la logique et les données) peut être définie. Pour être totalement implémentables aussi bien pour les nouvelles applications *Web* et ceux existantes, ces interfaces doivent être utilisées par différents langages de programmation.

2. Sécurité des communications interactives entre les clients *Web* et les serveurs. Ceci exige que la connexion à travers le *Web* entre un client et les services soit une communication dans les deux sens. La communication ne doit pas nécessairement passer par un serveur HTTP.

3. Installation du côté client. Dans cette étape on installe un environnement *Web* (typiquement un N.C.J) où de nouvelles applications et des *upgrades* peuvent être distribués sur demande et installés.

Les applets JAVA et le client/serveur

Avec l'introduction de JAVA on a rendu le *Web* interactif. Mais JAVA tout seul ne permet pas de faire du vrai client/serveur sur le *Web*.

Mais en combinant JAVA avec deux autres technologies on peut rendre le *Web* totalement client/serveur en suivant la méthodologie présentée en haut (Construction d'un système client/serveur *Web*). Ces deux technologies sont :

1. **L'infrastructure NEO[1] de SunSoft pour un client/serveur à trois niveaux**. **NEO** est l'acronyme de **NE**tworked **O**bjects. NEO est conforme à CORBA et permet l'accès aux objets et aux services qu'ils diffusent n'importe où sur le réseau. NEO implémente la logique de l'application (exemple : les applications de gestion, les applications de facturation, ...) et les données.

[1] http://www.sun.com/sunsoft/neo/index.html

2. **Joe**[1] **de SunSoft**. **Joe** est l'acronyme de *JAVA objects everywhere*. Joe est un **O**bject **R**equest **B**roker (ORB) qui connecte un *Applet* JAVA aux objets NEO distants s'exécutant sur n'importe quelle machine à travers Internet ou un Intranet. L'ORB Joe est automatiquement téléchargé dans le client *Web* avec les *Applets* JAVA. Joe établis ensuite et gère les connexions entre les objets JAVA locaux (*Applets*) et les objets NEO distants. Joe inclus aussi un compilateur **IDL-JAVA** qui génère automatiquement les classes JAVA associées aux définitions des interfaces des objets NEO qui sont écrites en IDL.

La figure 5.11 montre les étapes à suivre pour faire du développement avec Joe.

Etape 1. Obtenir une interface vers l'objet CORBA distant. Cette interface est évidemment écrite en IDL et on exécute le compilateur IDL-JAVA pour générer l'interface JAVA correspondante à l'objet distant. Ce compilateur génère un *stub* qui garantie une communication transparente avec les objets NEO.

Etape 2. Écrire le code client JAVA en utilisant d'une part l'interface JAVA générée (Etape 1) à l'objet distant et d'autre part Joe pour accéder à l'objet distant.

Etape 3. Compiler l'application ou l'*Applet* avec le compilateur JAVA (**javac**).

Etape 4. Exécuter le code.

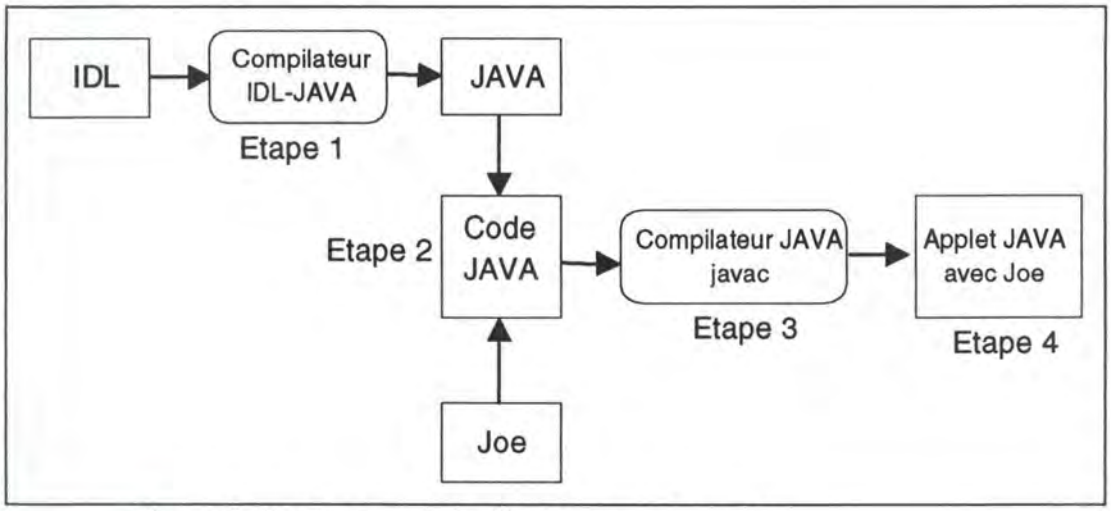


Figure 5.11 : Processus de développement avec Joe

L'avantage de l'approche Joe

Les applications *Web* créées en utilisant Joe, NEO et JAVA possèdent les avantages suivants :

1. En utilisant Joe et NEO les applications existantes (legacy ou autres) peuvent être disponibles sur le *Web* sans les réécrire.

[1] http://www.sun.com/sunsoft/neo/external/prod_specs/JOE_Overview.html

2. Une fois que les *Applets* JAVA clients sont chargés dans le contexte du navigateur client (figure 5.12 Etape 1), Joe permet une communication directe avec les objets distants sans passer par le serveur HTTP (figure 5.12 Etape 2).
Le serveur HTTP est alors utilisé exclusivement comme un distributeur de pages HTML et des *Applets* JAVA alors que les services NEO gèrent le traitement associé aux applications et l'interaction entre le client et le serveur distant abritant ces services NEO (figure 5.12 Etape 2).

3. Avec Joe et NEO les applications *Web* peuvent être hautement interactives en accédant
dynamiquement aux services réseau. A la différence des formulaires HTML et des scripts CGI qui génèrent et transfèrent de nouvelles pages *Web* chaque fois qu'une requête est lancée, les *Applets* JAVA avec Joe envoient seulement l'information mise-à-jour fournissant de ce fait des réponses instantanées avec un trafic réseau faible.

4. Les *Applets* JAVA s'exécutant dans le contexte du navigateur du client *Web* gèrent la présentation et l'interaction avec l'utilisateur alors que Joe permet à l'*Applet* téléchargé via le *Web* en utilisant le protocole HTTP d'accéder à d'autres services sur d'autres serveurs utilisant d'autres protocoles garantissant la notion de **session** et de **transaction** entre le client et le serveur.

5. Joe fournis une session complète client/serveur. Joe offre non seulement l'accès aux objets distants à partir d'un programme JAVA (client vers le serveur) mais fournit aussi l'accès de l'objet distant vers le programme JAVA (serveur vers client).
Les communications du serveur vers le client sont très utiles dans les applications où les mises-à-jour asynchrones (tel les interruptions) doivent être fournies à l'utilisateur.

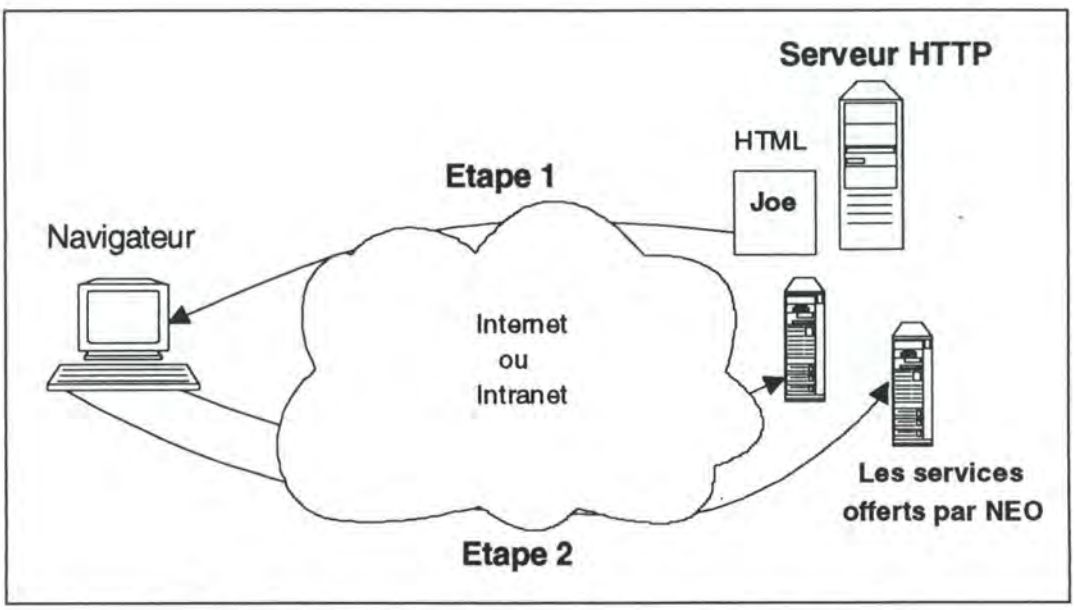


Figure 5.12 : Utilisation d'un serveur HTTP et de NEO

5.10 Le modèle Objets Distribués utilisant JAVA

Dans ce paragraphe on présente le modèle objets distribués utilisant JAVA.

Dans le modèle objets distribués JAVA, un **objet distant** est celui dont les méthodes peuvent être invoquées à partir d'une autre **Machine Virtuelle JAVA**, potentiellement sur un *host* distant. Un objet de ce type est décrit par une ou plusieurs **interfaces** distantes qui sont des **Interfaces** du langage JAVA (5.4 notion d'Interface) déclarant les méthodes de l'objet distant.

Pour les mécanismes de communication de base, JAVA supporte les *sockets*. Mais les sockets obligent le client et le serveur à encoder et à décoder les messages d'échange. Un mécanisme d'invocation de méthode à distance est donc nécessaire. Dans le chapitre 4 on a présenté deux modèles celui de CORBA(ORB) et celui de OLE(COM/DCOM).

Dans le monde JAVA on parle de **RMI** (***Remote Method Invocation***) pour définir la façon avec laquelle un objet JAVA peut invoquer la méthode d'un autre objet JAVA sur le même host ou sur un host distant. Dans de tel système un objet local "subrogé" (*stub*) gère les invocations aux objets distants.

Le Remote Method Invocation (**RMI**[1])

Le système **RMI** est composé de 3 couches (figure 5.13) : la couche ***stub/skeleton***, la couche **référence à distance** et la couche **transport**. La frontière de chaque couche est définie par un protocole et une interface spécifique, chaque couche est par conséquence indépendante par rapport aux autres et peut être remplacée par une autre implémentation sans affecter les autres couches du système.

Par exemple actuellement l'implémentation de la couche transport est basée sur le protocole TCP (en utilisants les sockets JAVA) mais une implémentation basée sur UDP peut être aussi utilisée.

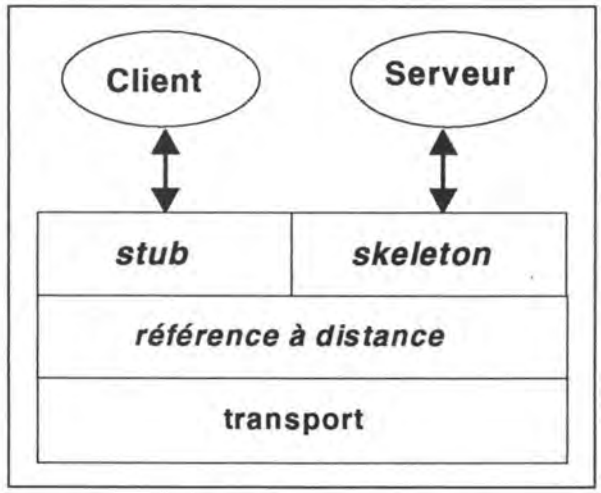


Figure 5.13 : Le système RMI

[1] http://chatsubo.javasoft.com/current/rmi/index.html

• **La Couche *stub/skeleton***

Le mécanisme est le même que pour les objets distribués étudiés dans le chapitre 4, en effet un client invoquant une méthode d'un objet serveur à distance utilise un *stub* (ou *proxy*) pour l'objet à distance comme une "conduite" vers l'objet distant. La couche *stub/skeleton* est l'interface entre la couche application et le reste du système RMI. Cette couche est indépendante par rapport à la couche transport et transmet donc les données à la couche de référence objet via une abstraction appelée **séquence de *marshaling***. Rappelons que le *marshaling* permet de spécifier la syntaxe des messages **in** et **out** passés entre le client et le serveur, le *marshaling* spécifie comment il faut passer les paramètres et le principe de codage. **La séquence de *marshaling*** emploie un mécanisme appelé ***object serialization*** qui permet aux objets JAVA d'être transmis entre les différents espaces d'adressage, l'*object serialization* définie des interfaces pour écrire et lire les objets et les classes qui implémentent ces interfaces. Les objets transmis en utilisant l'*object serialization* sont passés par copie à l'espace d'adressage distant à moins qu'ils ne soient des objets distants et dans ce cas ils sont passés par référence.

Le *stub* client implémente toutes les interfaces supportées par l'implémentation de l'objet distant.

Le *stub* client est donc responsable de :

1. L'initialisation d'un appel à l'objet distant (en appelant la couche référence à distance).

2. Faire du *marshaling* des arguments (c'est à dire faire du *packaging* "mise en paquet" des paramètres de l'invocation) vers une **séquence de *marshaling*** (obtenue de la couche référence à distance).

3. Informer la couche référence à distance que l'appel doit être invoqué.

4. Faire de l'*unmarshaling* (c'est à dire "extraire du paquet" les paramètres de l'invocation) de la valeur de retour ou de l'exception (si un problème survient). Ceci se fait en analysant la **séquence de *marshaling***.

5. Informer la couche référence à distance que l'appel est terminé.

Le *skeleton* serveur d'un objet distant contient une méthode responsable de la répartition des appels à l'implémentation effective de l'objet distant. Le *skeleton* est responsable de :

1. Faire de l'*unmarchaling* des arguments en analysant la **séquence de *marshaling***.
2. Faire un appel montant à l'implémentation effective de l'objet distant.
3. Faire du *marshaling* de la valeur de retour de l'appel ou une exception (si un problème survient) sur la **séquence de *marshaling***.

• **La couche *référence à distance***

La couche référence à distance est responsable de la sémantique de l'invocation. Cette couche est par exemple responsable de la détermination si le serveur est un seul objet ou un objet répliqué exigeant de ce fait des communications avec plusieurs emplacements. Chaque implémentation d'objet distant choisie sa propre sémantique de référence si le serveur est un seul objet ou au contraire c'est un objet répliqué exigeant donc plusieurs communications.

Plusieurs protocoles d'invocation peuvent être supportés dans cette couche par exemple :

1. Une invocation point-à-point *unicast*.
2. Invocation à des groupes d'objets répliqués.
3. Supporter une stratégie de réplication spécifique.
4. Supporter une référence persistante à un objet distant (permettant de ce fait l'activation de l'objet distant).
5. Stratégies de reconnexion (si l'objet distant devient inaccessible).

La couche référence à distance possède deux composants qui coopèrent entre eux : un **composant côté client** et un **composant côté serveur**. Le composant client contient des informations spécifiques au serveur distant (ou aux serveurs distants si les références distantes sont pour des objets répliqués) et communique avec le composant serveur via la couche **transport**.

Durant chaque invocation de méthode les composants clients et serveurs garantissent la sémantique des références à distance. Par exemple si un objet distant fait partie d'un objet répliqué, le composant client fait suivre l'invocation à chaque objet répliqué.

Le **composant côté serveur** implémente la sémantique des références à distance avant de délivrer une invocation de méthode à distance au *skeleton*. Ce composant peut par exemple prendre en charge l'atomicité des différentes communications avec les autres serveurs.

• **La couche transport**

La couche transport établis la connexion, gère la connexion, garde trace et distribue les invocations aux objets distants (les cibles des appels distants) résidant dans l'espace d'adressage de transport.

La couche transport fait suivre l'appel distant à la couche référence à distance. La couche référence à distance prend en charge le comportement côté serveur nécessaire avant de transmettre la requête au *skeleton* serveur. Le *skeleton* d'un objet distant fait un appel à l'implémentation de l'objet distant qui prend en charge l'appel.

La valeur de retour de l'appel est envoyée via le *skeleton*, la couche référence à distance et le transport du côté serveur, et ensuite transmise via le transport, la couche référence à distance et le *stub* côté client.

En général la couche transport du système RMI est responsable de :

1. L'établissement des connexions aux espaces d'adressage distants.
2. La gestion des connexions.
3. L'écoute des appels entrants.
4. La maintenance d'une table des objets distants qui résident dans l'espace d'adressage.
5. L'établissement d'une connexion pour les appels entrants.
6. La localisation du distributeur qui gère les cibles des appels distants et faire passer la connexion à ce distributeur.

La représentation concrète d'une référence objet distant consiste en un **point d'arrivée** et un **identificateur d'objet**. Cette représentation porte le nom de **référence vivante**. Étant donnée une référence vivante d'un objet distant, un transport peut utiliser un **point d'arrivée** afin d'établir une connexion à l'espace d'adressage dans lequel l'objet distant réside. Du côté serveur, le transport utilise l'**identificateur d'objet** pour localiser la cible de l'appel distant.

Le transport du système RMI est constitué de 4 abstractions de base :

1. Un **point d'arrivée** qui est une abstraction utilisée pour dénoter un **espace d'adressage** ou la **Machine Virtuelle JAVA**. Dans l'implémentation un point d'arrivée peut être traduit à son transport. C'est à dire étant donné un point d'arrivée une instance de transport spécifique peut être obtenue.

2. Un **canal** est une abstraction désignant une conduite entre deux espaces d'adressage. Il est responsable de la gestion des connexions entre un **espace d'adressage local** et un **espace d'adressage distant**.

3. Une **connexion** qui est une abstraction pour le transfert des données (exécutant des entrées/sorties).

4. Le transport est une abstraction qui gère les canaux. Chaque canal est une connexion virtuelle entre deux espaces d'adressage. Dans un transport, seulement un seul canal existe pour une paire d'espace d'adressage : l'espace d'adressage local et l'espace d'adressage distant. Étant donné un **point d'arrivée** à un espace d'adressage distant, un transport établit un **canal** à cet espace d'adressage. L'abstraction transport est responsable aussi de l'acceptation des appels sur des connexions entrantes à l'espace d'adressage, de l'établissement d'une connexion objet pour l'appel et la distribution aux couches de haut niveau du système.

Un transport défini la représentation concrète d'un **point d'arrivée**, de telle façon que plusieurs implémentation transport puissent exister. La conception et l'implémentation supportent aussi plusieurs transports par espace d'adressage, donc aussi bien TCP qu'UDP peuvent être supportés dans la même machine virtuelle.

5.11 Conclusion

JAVA est surtout utilisé pour inclure de l'animation et de interactivité dans les pages HTML. Inclure du code compilé en format Bytes-Code (neutre et portable), pouvoir télécharger ce morceau de code et l'exécuter localement dans l'environnement du client *Web* telle est l'idée derrière JAVA. On a vu aussi que combiner JAVA avec d'autres technologies : Joe et RMI permet de faire du vrai client/serveur sur le *Web* sans passer obligatoirement par le protocole HTTP.

Mais deux défis majeurs existent : D'une part la sécurité du code téléchargé, certes assuré par les 4 niveaux de sécurité que l'on a présenté, mais si on désire utiliser d'autres serveurs que le serveur HTTP la sécurité n'est pas totalement garantie et d'autre part le débit transactionnel qui est faible puisque JAVA est un langage interprété et donc ne permet pas des traitements transactionnels performants lors de l'exécution.

Chapitre 6 : LE *WHITEBOARD*

6.1 Introduction

Le ***Whiteboard***[1] est un outil de haut niveau du **bloc multimédia** de la plate-forme **PRISM** (Chapitre 2). Sa fonction est de fournir un espace de travail commun pour un ensemble d'utilisateurs.

Le *Whiteboard* permet un échange d'information d'un poste ***master*** vers un ensemble de poste en attente d'information appelé ***slave***.

La réalisation du *Whiteboard* a été divisée en 2 étapes. Premièrement le développement d'une version *stand alone* du *Whiteboard*, pour intégrer les fonctions de dessin et pour tester de nouvelles caractéristiques. La seconde étape gère les aspects de communication du *Whiteboard*, y compris l'échange de données de dessin entre le conférencier (en mode *master*) et les participants (en mode *slave*) et la possibilité d'activer ou de désactiver le mode *master* en mode *slave*.

6.2 Communication avec la plate-forme PRISM

Le *Whiteboard* communique en utilisant les commandes PRISM implémentées dans le système d'exploitation UNIX. Le nommage et l'appel de ces fonctions PRISM vont être restreints aux noms des commandes de programmation de communications qui existent déjà.

6.3 Graphiques importés

Tous les slides utilisés dans l'application *Whiteboard* sont préparés à l'avance. Le conférencier convertit ses graphiques avec les procédures *shell* fournies au format utilisé dans le *Whiteboard*. On utilise le format *Graphical Interchange Format* (GIF) 128 couleurs. Ce codage permet une compression raisonnable des données images et un accès rapide pour le décodage et l'affichage des images à l'écran.

---

[1] [Hub94] p: 13-30

6.4 Modes d'opération

Dans ce paragraphe, on va voir les fonctionnalités offertes par le *Whiteboard*.

Le *Whiteboard* doit supporter deux modes d'opération (figure 6.1).

Un mode ***master*** dans lequel toutes les fonctions de dessin sont disponibles et qui va diffuser les informations de dessin sur le réseau, ce mode est propre au conférencier.

Pour recevoir les informations de dessin, un mode ***slave*** est nécessaire. Les deux modes utilisent les mêmes fonctions de dessin pour afficher l'information sur l'écran.

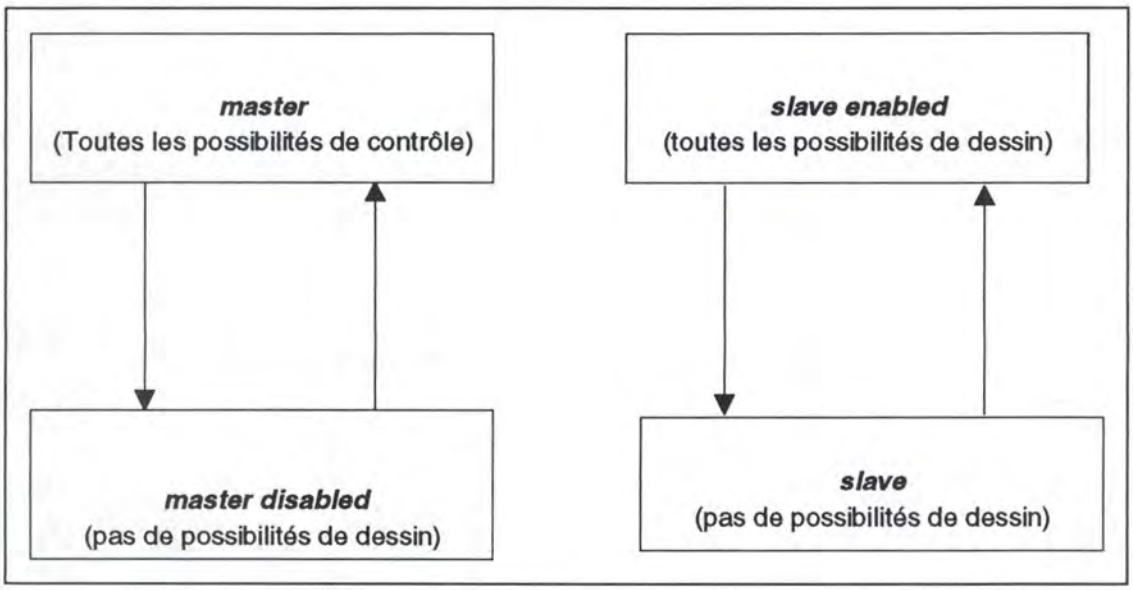


Figure 6.1 : Les modes d'exécution du *Whiteboard*

Dans la figure 6.1 on voit les passages entre les différents modes de l'application. On peut passer par exemple du mode *master* avec toutes les possibilités de dessin vers le mode *slave* où aucune possibilité de dessin n'est permise.

6.4.1 Mode *slave*

Le *Whiteboard* en mode *slave* possède les deux fonctionnalités suivantes : l'affichage et l'écoute.

La phase d'initialisation est divisée en deux parties :

. La création de la fenêtre principale sur laquelle on va dessiner.
. La création d'une entrée à partir de laquelle les informations de dessin vont être lues.

Après cette phase d'initialisation, le *Whiteboard* dans le mode *slave* attend l'arrivée des paquets, les lis, décode le message et exécute le dessin.

6.4.2 Mode *master*

Dans le mode *master*, des outils sont crées pour dessiner dans la fenêtre principale. Actuellement ce mode se déclenche directement dans la phase d'initialisation. Il prévoit de développer un *Accesscontroler* pour les Services Multimédia de Haut Niveau, qui peut activer ou désactiver le mode master. L'application *Whiteboard* fait déjà cette fragmentation, pour permettre ces traitements.

Fenêtre des outils de dessin

Pour dessiner dans la fenêtre principale, plusieurs fonctionnalités sont disponibles. Mettre du texte, dessiner un cercle, une ligne, un rectangle, une main levée sont les fonctions de base, qui sont utilisées dans la plupart des programmes de dessin. Les accès sont accomplis en : enfonçant les boutons qui doivent être cliqués par la souris.

Fenêtre des outils de la souris

Pour indiquer un endroit bien déterminé de la fenêtre principale 3 types de pointeurs sont utilisés :
. Une souris fixe placée en cliquant de façon explicite sur le bouton de la souris là où le pointeur doit être placé.
. Une souris mobile qui suit le pointeur réel à l'intérieur de la fenêtre principale.
. Une souris cachée qui n'est pas réellement un pointeur mais un marqueur. Il marque quelle partie du slide doit être montrée et quelle partie doit être cachée.

Un ascenseur d'icônes pour le contrôle des slides

Un discours est normalement prononcé en suivant l'ordre des slides que le conférencier a préparé à l'avance. Mais puisque le *Whiteboard* doit permettre aux participants de poser des questions, un outil est donc nécessaire pour accéder aux slides qui ne sont pas dans l'ordre dans la conférence.

Chaque slide est aussi représenté par une icône. L'ascenseur permet à l'utilisateur de naviguer à travers les icônes pour trouver rapidement le slide voulu.

6.5 Interaction réseau

L'application doit prendre en charge deux types d'entrée (figure 6.2) :

1. Une entrée directe du conférencier qui travaille avec le *Whiteboard* et l'ajout de changement comme du texte supplémentaire, du dessin, etc...
2. Une entrée indirecte délivrée par le réseau à l'application

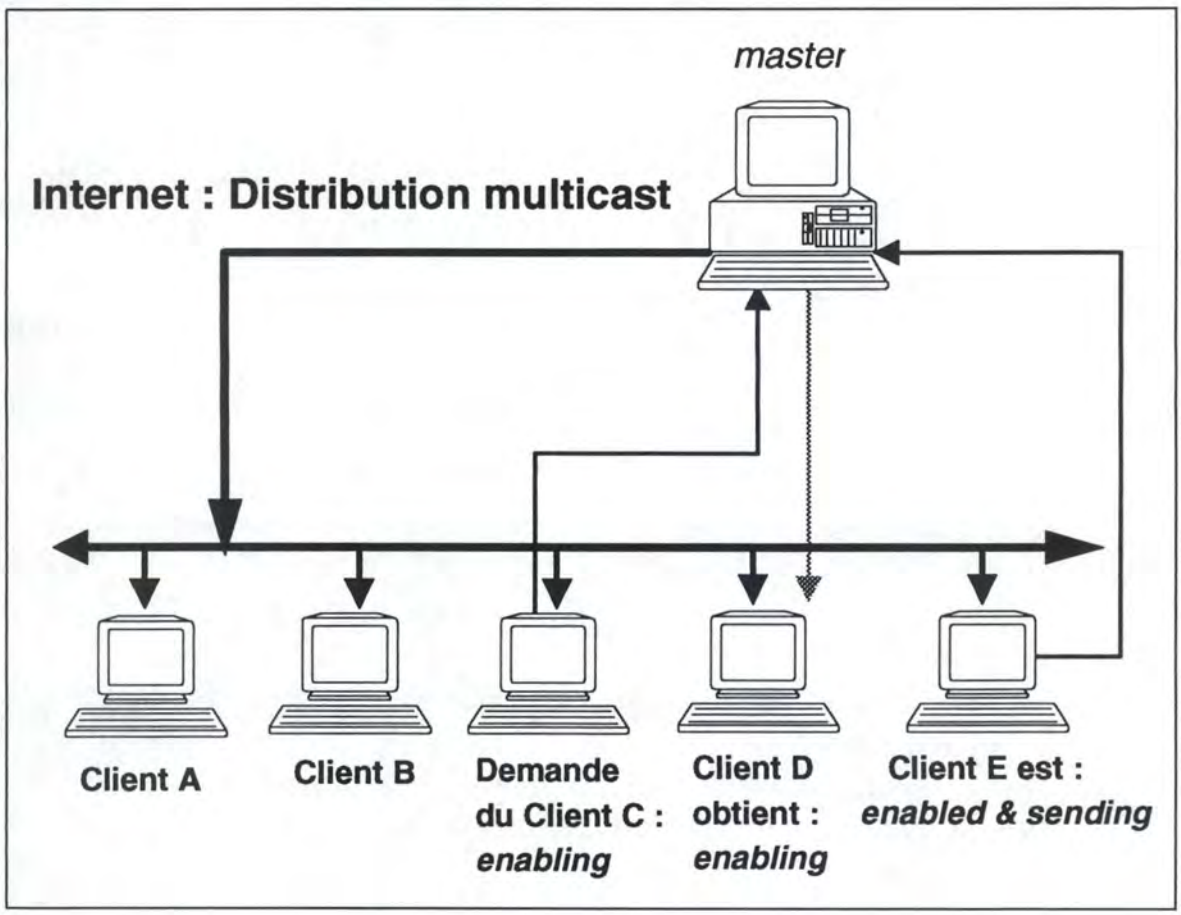


Figure 6.2 : Interaction réseau

Les modifications en provenance du conférencier doivent être affichées là où les autres participants se trouvent.

La différence entre un ***slave*** **actif** (le cas d'un auditeur par exemple qui désire intervenir) et un ***master*** n'est pas visible du point de vue du *Whiteboard*. C'est une vue externe qui conduit à cette distinction, en considérant le type de la connexion utilisé pour transmettre les données.

Seulement le master de la conférence détient le contrôle sur les **adresses *multicast*** donc si les participants veulent diffuser quelque chose ils doivent utiliser une connexion point-à-point avec le master de la conférence. Le conférencier envoie donc la donnée à tous les participants via la liaison *multicast*. Le *master* peut aussi déconnecter une entrée ***slave***, même si ses outils de dessin sont inactifs.

La même différence doit être faite entre un *master* inactif et un *slave*. Le *master* inactif peut retourner de sa propre initiative et contrôler la session. Il ne demande pas d'être activé par le *slave*, qui possède actuellement le contrôle. Au contraire, le *slave* doit attendre une autorisation du master pour avoir les outils de dessin.
Une vue externe peut donner au *master* de la conférence le droit de redirectionner ou de diffuser l'information dessinée, alors que le *slave* peut seulement délivrer l'information dessinée au master dans une connexion point-à-point directe.

6.5.1 Format des paquets

Les messages distribués par le *Whiteboard* possèdent différentes caractéristiques et peuvent transporter plus ou moins de données. Comme conséquence la donnée contenue dans un message possède différentes représentations, suivant l'action exécutée. Pour décoder les messages, il est utile d'utiliser une seule en-tête, qui représentera tous les paquets transportés.

Comme les messages n'ont pas la même taille, le décodage est formé de deux phases :

. Phase 1 : Décodage de l'en-tête pour déterminer l'action
. Phase 2 : Suivant l'action, décodage du sous-paquet

L'en-tête du paquet

L'en-tête du paquet est composé de trois champs (figure 6.3) :

. Un champ ***timestamp***, utilisé dans l'application pour les statistiques et la synchronisation
. Un champ **action** qui représente un code de commande assigné
. Un champ **paramètre** qui accompagne l'action
Dans la version actuelle du *Whiteboard*, le *timestamp* est utilisé pour des données statiques concernant la qualité de transport à travers le réseau. Plusieurs mesures statistiques, qui peuvent être déjà intégrées dans le code du *Whiteboard*, peuvent être appliquées.

L'**action** est un code pour une commande particulière exécutée par le master. Elle correspond à une fonction de dessin, qui doit être exécutée. Le *master* et le *slave* utilisent les mêmes fonctions de dessin pour modifier l'écran.

Le champ **paramètre** est utilisé pour transmettre la couleur de l'action de dessin. Pour certaines commandes comme l'affichage d'un slide, le champ n'est pas utilisé. Mais pour que de nouvelles fonctionnalités puisse être ajoutées au *Whiteboard* pour améliorer son utilisation, ce champ peut être très utile. Il peut être utilisé par exemple pour dessiner des objets à trois dimensions, donc la fonctionnalité de dessin peut attendre avant de placer l'objet sur l'écran, jusqu'à ce que toutes les coordonnées ou que les paramètres de l'objet arrivent. Le paramètre peut indiquer si un autre paquet va survenir, et qui contient les autres parties de l'objet.

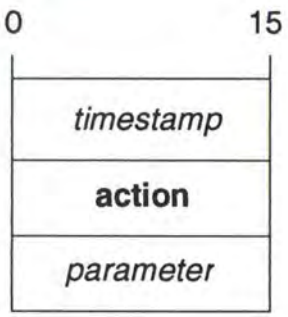


Figure 6.3 : Format de l'en-tête du paquet

Le sous-paquet contenant une seule paire de coordonnée

Pour plusieurs fonctionnalités de dessin une seule paire de coordonnées est suffisante pour exécuter une action de dessin. Le pointeur de la souris par exemple nécessite seulement une seule paire de coordonnées (figure 6.4).

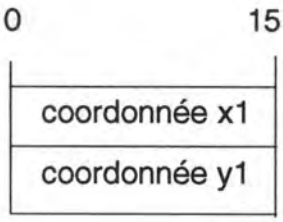


Figure 6.4 : Format du sous-paquet à une seule paire de coordonnée

Le sous-paquet contenant une double paire de coordonnées

Pour d'autre fonctionnalités graphiques une double paire de coordonnées (figure 6.5) sont nécessaires pour placer le graphique dans la surface de dessin (tracer une ligne par exemple).

L'utilisation de deux paquets avec chacun une paire de coordonnées peut aussi accomplir le dessin mais il y a beaucoup d'**overhead** transporté par chaque paquet, donc la production de trafic supplémentaire sur le réseau doit être évitée.

Les raisons principales d'avoir un seul sous-paquet contenant une double paire de coordonnées au lieu de deux sous-paquet contenant une seule paire de coordonnée chacun sont :

. L'en-tête du Protocole de Transport PRISM (PTP) et l'en-tête du datagram IP vont causer des overhead supplémentaires.
. La position définitive d'un rectangle est transmise quand il est complètement dessiné par le master de la conférence. Et donc les deux paires de coordonnées sont valides au même moment.
. L'application n'a pas besoin de prendre en considération les problèmes de transmission et donc des problèmes de cohérence tels que :

. La première paire de coordonnées est arrivée après la deuxième paire.
. Une paire de coordonnées est perdue.

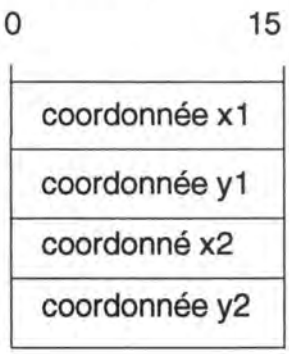


Figure 6.5 : Format du sous-paquet contenant une double paire de coordonnée

Le sous-paquet contenant n'importe quelle donnée

Le champ **Donnée** (figure 6.6) désigne soit un message qu'il faut afficher à l'écran soit le nom d'un slide qu'il faut alors charger. Une paire de coordonnées est utilisée pour placer le message ou le slide à l'écran.

Figure 6.6 : Format du sous-paquet contenant n'importe quelle donnée

6.5.2 Gestion des événements

Si l'application se comporte comme un programme de dessin normal, chaque action de dessin est alors immédiatement exécutée et aucune contrainte sur une transmission ne doit être tenue en compte. Mais le *Whiteboard* est utilisé pour transmettre les dessins locaux aux participants, qui recevrons les mêmes dessins que le conférencier qui les a crée sur son écran.

Ceci a pour conséquence que l'exécution des modifications dans la fenêtre principale doit être faite en deux phases (figure 6.7) :

. La **phase de prétraitement** dans laquelle les événements sont analysés et transmis au réseau.

. La **phase d'exécution** dans laquelle le dessin est définitivement exécuté.

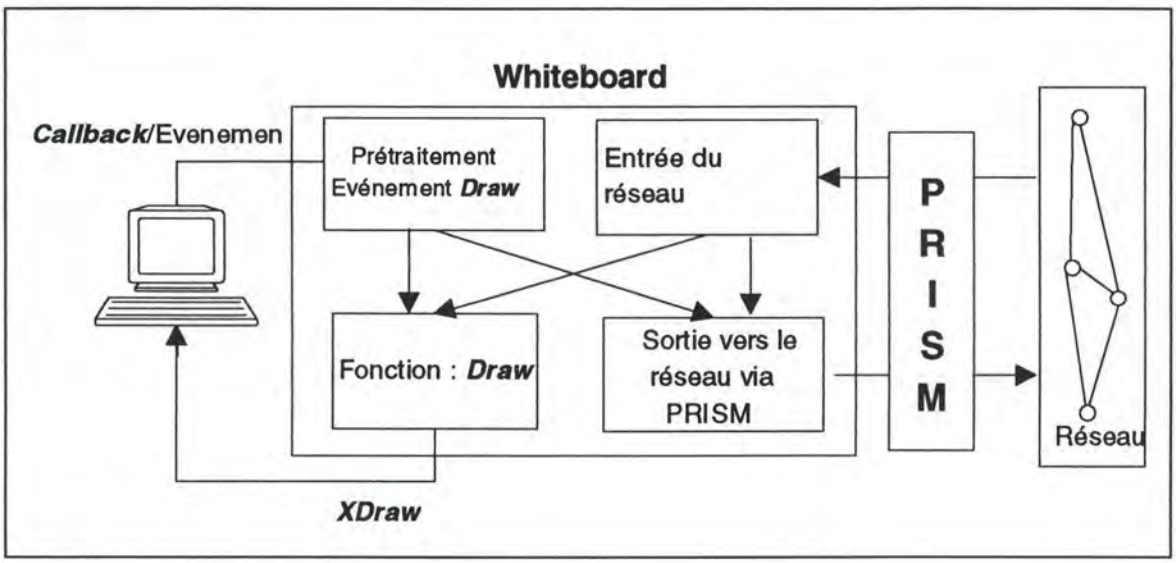


Figure 6.7 : Gestion des événements

La phase de prétraitement fournit aussi des indications de dessin pour le *master* de la conférence, qui exécute le dessin, comme le fait de mettre en ombre la forme à dessiner. Quand le dessin est complet, l'information est transmise sur le réseau et sera ensuite dessinée à l'écran.

6.6 Conclusion

Le *Whiteboard* est un service de haut niveau du bloc multimédia de la plate-forme PRISM. Le *Whiteboard* offre des outils d'interface pour réaliser des fonctions de dessin. Le Whiteboard peut être activé selon 2 modes. Le mode *master* qui contrôle les outils de dessin et crée les dessins. Le mode *slave* qui permet la réception des modifications faite par le conférencier.

Le mode *slave* peut aussi prendre l'initiative et réaliser des dessins auquel cas une demande préalable doit être formulé au mode *master*.

Chapitre 7 : Les deux *Applets* JAVA réalisés

7.1 Introduction

Notre participation au projet PRISM a été de porter le mode ***slave*** **passif** (c'est-à-dire le cas de l'auditeur qui visualise les actions du conférencier et entend son discours mais sans possibilité d'intervention) de l'application ***Whiteboard*** sur le ***Web***.

Donc par rapport à la plate-forme PRISM on a travaillé sur le PRISM FILE MGT (pour gérer les cours enregistrés). La partie distribution est prise en compte par les *Applets* JAVA. Les fonctionnalités du bloc de gestion ne sont pas utilisées.

En utilisant JAVA à travers le concept d'*Applet* (5.5) n'importe quel utilisateur doté d'un Navigateur Conforme à JAVA (**N.C.J**) peut charger une page HTML à travers le *Web* et suivre le cours préenregistré. De ce fait on permet à la page HTML de faire du traitement local, de l'animation et de la synchronisation des tâches qui sont nécessaires afin de réaliser le mode *slave* passif de l'application *WHITEBOARD*. Néanmoins vu les restrictions de sécurité imposées par la plupart des N.C.J et notamment Netscape (5.7.4) les deux applets développées s'exécutent seulement avec l'outil de visualisation d'*Applet* ***appletviewer*** fournis par les laboratoires Sun. L'outil ***appletviewer*** se comporte de la même façon qu'un N.C.J c'est à dire qu'il prend comme argument une URL et demande au client Web de télécharger la page HTML correspondante à cette URL. Les deux *applets* vont alors s'exécuter de façon normale dans le contexte de la page HTML, mais *appletviewer* à la différence des N.C.Js permet de définir une politique de sécurité propre en autorisant par exemple l'écriture ou la lecture de certains fichiers bien déterminés sur le *host*.

JAVA offre aussi la possibilité d'avoir une interface universelle (figure 7.1) du fait que le développeur manipule des outils d'interface abstraite et son caractère de portabilité et d'architecture neutre (5.4 Architecture neutre, Portabilité et Robustesse) permet à l'application de tourner sur n'importe quel système d'exploitation et sur n'importe quel processeur. Ceci est d'une grande utilité pour implémenter le mode *slave* passif, l'utilisateur ne change rien à l'application il peut disposer d'un environnement Mac, Windows ou X-windows le résultat est toujours le même.

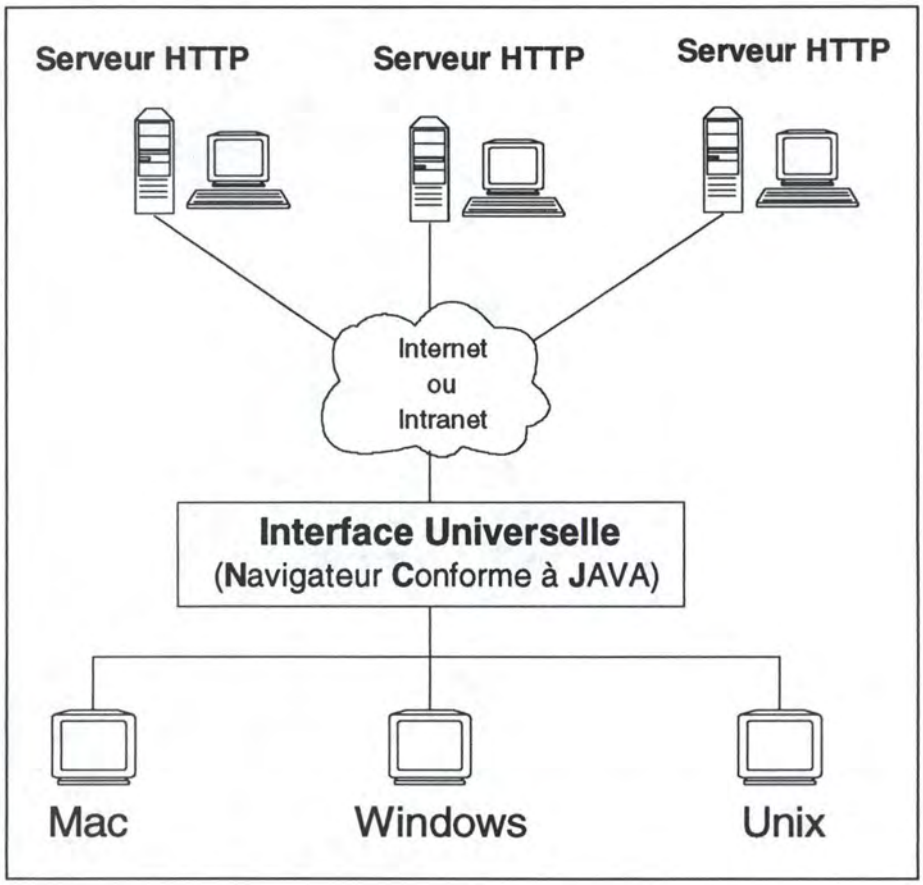


Figure 7.1 : Interface Universelle

7.2 Les pages HTML des deux *Applets*

On a réalisé deux *Applets* :

1. Le premier *Applet* **Reproduction_Action** correspond à la seule reproduction des actions du conférencier sur l'écran c'est-à-dire l'**affichage des slides** (images) et les **actions effectuées** sur ces slides sans reproduire le discours du conférencier.

2. Le deuxième *Applet* **Reproduction_Action_Discours** correspond à la **reproduction** aussi bien des **actions** du conférencier que de son **discours**.

• Voici la page HTML qui correspond à l'*Applet* **Reproduction_Action** :

<title> Mode Slave Passif </title>

<hr>

<*Applet* code = "Slave1.class" **width** = 800 **height** = 800>
<param name = fichier_images **value** = "/bocage.données">
</Applet>

</hr>

Le Slave1.class est généré par la compilation du code de l'*Applet* :
Reproduction_Action.

Le fichier "/bocage.données" est un fichier de test que le navigateur passe comme paramètre à l'*Applet*.

• Voici la page html qui correspond à l'*Applet* **Reproduction_Action_Discours** :

<title> Mode Slave Passif </title>

<hr>

<*Applet* code = "Slave2.class" **width** = 800 **height** = 800>
<param name = fichier_images **value** = "/bocage.donnees">
<param name = fichier_son **value** = "/bocage.francais">
</Applet>

</hr>

Le Slave2.class est généré par la compilation du code de l'*Applet* :
Reproduction_Action_Discours

Les fichiers "/bocage.données" et "/bocage.français" sont les deux fichiers de test que le navigateur passe comme paramètres à l'*Applet*.

Le développement des deux *Applets* est constitué de 3 phases (figure 7.2).

Les phases 4 et 5 représentent le chargement de la page HTML et la visualisation de l'exécution de l'*Applet* par un client *Web* utilisant un N.C.J.

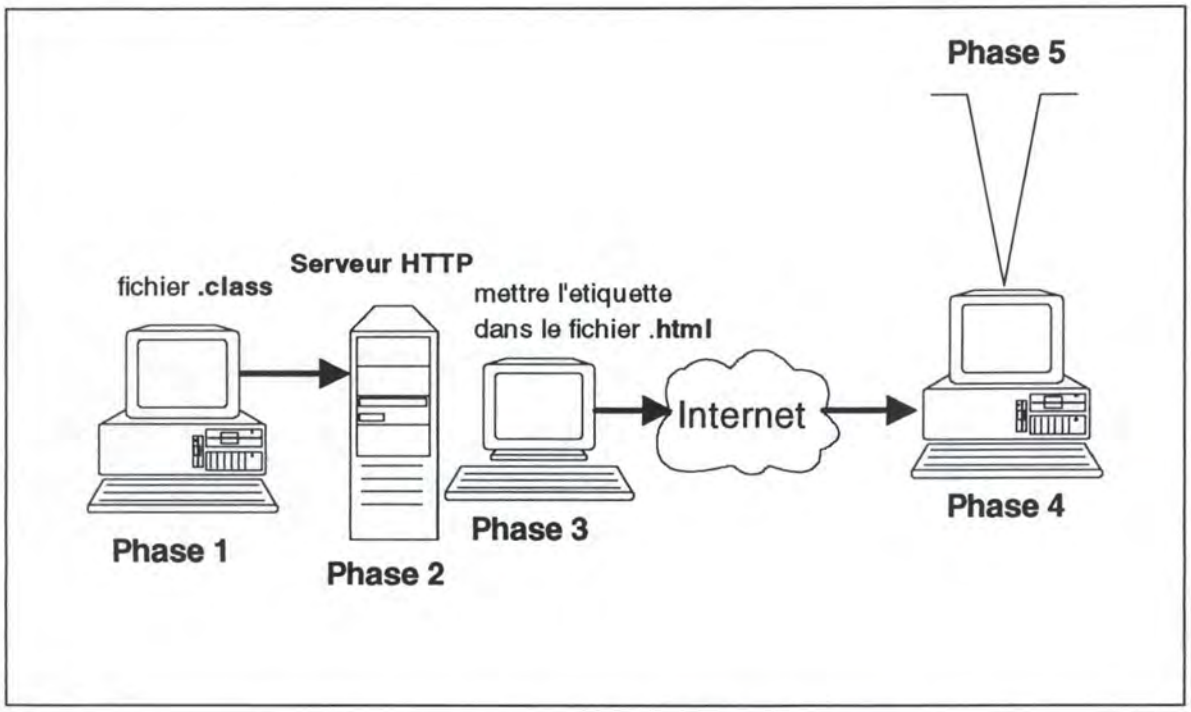


Figure 7.2 : Insertion d'un *Applet* dans une page HTML

Phase 1 : Dans cette phase, on compile le code Slave1.java (Slave2.java) avec la commande **javac** et on obtient un fichier en format Bytes-Code Slave1.class (Slave2.class) qui ne contient donc aucuns détails d'implémentation.

Phase 2 : On met le fichier Slave1.class (Slave2.class) sur le serveur HTTP dans le même répertoire que celui qui contient les pages HTML ou dans l'un des sous-répertoires.

Phase 3 : Dans la page HTML on met l'étiquette *<Applet ... /Applet>* comme indiquée dans les deux pages HTML précédentes.

Les phases 1, 2, 3 représentent la mise de la page HTML (et donc de l'*Applet*) à la disposition des clients sur le *Web*.

Phase 4 : Le N.C.J lit la page HTML, rencontre l'étiquette *<Applet ... /Applet>* et génère la requête HTTP ***get*** pour télécharger le fichier Slave1.class (Slave2.class) dont l'url est la même que celle de la page HTML.

Phase 5 : Le N.C.J crée une nouvelle fenêtre, lance un nouveau thread et exécute le code téléchargé.

Il faut noter que vu que c'est un cours préenregistré, l'utilisateur ne peut pas intervenir et de ce fait le mode *slave* passif est maintenu durant toute la durée de l'exécution de l'*Applet* sans possibilité de transition vers le mode *master*.

Dans les deux dernières parties, on va présenter les deux *Applets* réalisés.

Il y a des phases qui sont communes aux deux *Applets*, notamment les 3 premières phases (phase 1, phase 2 et phase 3) qui correspondent conceptuellement aux préparatifs avant l'intervention du conférencier.

Ces préparatifs dans une conférence dans le monde réel concernent la distribution des slides aux participants avant que le conférencier ne puisse intervenir.

Pour simuler le mode *slave* les deux fichiers de test (c'est-à-dire le fichier_images représentant les actions de l'utilisateur et le fichier_son représentant le discours du conférencier) sont formés d'une suite de paquets. Chaque paquet est constitué de 9 champs (figure 7.3).

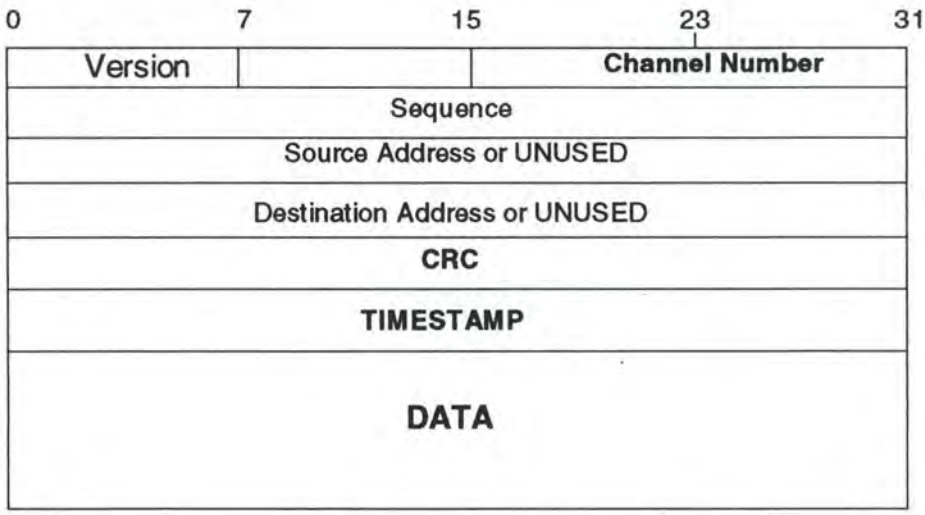


Figure 7.3 : Format des fichiers en entrées

Parmi les champs importants on a le champ ***Channel Number*** qui permet de spécifier le genre du paquet : Est ce que c'est un paquet ***Whiteboard***, un paquet **son** ou un autre paquet circulant sur le réseau.

Les 8 premiers champs ont une longueur fixe, le champ **DATA** est de longueur variable puisque c'est l'application *Whiteboard* qui génère ce champ (6.5.1).

Si le ***Channel Number*** est à 2 alors c'est un paquet *Whiteboard*. Dans ce cas on le décode. Le décodage consiste à extraire le champ **CRC** qui représente la longueur du paquet et de ce fait permet de savoir la suite des actions à exécuter, puisque à chaque paquet est associé une action. On lit aussi le champ **TIMESTAMP** qui représente le moment ou il faut exécuter l'action. Ce champ est très important pour réaliser la synchronisation des actions.

Le champ **DATA** représente la donnée générée par l'application *Whiteboard* (6.5.1).

Si le *Channel Number* est à 1 alors c'est un paquet son et non un paquet *Whiteboard*. Comme dans le premier cas on lit le CRC, TIMESTAMP et on lance les données son c'est-à-dire le champ DATA sur la sortie audio.

Si le *Channel Number* prend une valeur autre que 1 ou 2 on décode le champ **CRC** pour savoir la longueur du paquet et on lit le reste du paquet sans décodage.

7.3 L'*Applet* Reproduction_Action

La figure 7.4 montre les différentes phases par lesquelles passe l'*Applet* : **Reproduction_Action**.

Phase 1 : Dans cette première phase l'*Applet* initie une communication avec le navigateur qui représente son environnement d'exécution. Cette communication se traduit par la récupération du paramètre fournit par le navigateur et dont la valeur représente le nom du fichier en entrée à savoir le fichier des actions.

Cette communication se fait en mettant dans la page HTML, l'étiquette suivante :

< **param name** = fichier_images **value** = "/bocage.données" >

Ici la page HTML met à la disposition de l'*Applet* qui s'exécute dans son contexte un paramètre à savoir fichier_images dont la valeur est bocage.données le nom du fichier de test. Ce fichier représente les actions du conférencier que l'on doit reproduire sur l'écran de l'utilisateur.

L'*Applet* récupère dans le code ce paramètre en invoquant la méthode suivante :

getParameter("fichier_images")

Cette méthode renvoie à l'intérieur de l'*Applet* la valeur : /bocage.données.

Phase 2 : La phase 2 a pour but de télécharger du serveur le fichier : /bocage.données et d'en extraire les noms des slides qui vont être utilisés par le conférencier.

Pour cela on manipule une classe de haut niveau, à savoir la classe **URL**. URL est l'acronyme de Uniform Ressource Locator qui désigne une référence (ou une adresse d'une ressource sur le *Web*).

Cette phase comporte 3 étapes :

Etape 1. Ouverture d'une connexion vers le serveur en exécutant la méthode suivante :
new (**"http"**, "www.info.fundp.ac.be","/bocage.données"), cette méthode renvoie une instance url_image de la classe **URL** qui est une classe fournit par JAVA pour ouvrir des connexions de haut niveau.

http représente le protocole HTTP qui dans le monde Internet a pour but de gérer l'échange d'une requête et d'une réponse entre un client et un serveur. "www.info.fundp.ac.be" représente le host d'origine (serveur) auquel on se connecte et qui fournit la page HTML et donc l'*Applet*.

/bocage.données représente le nom du fichier des actions exécutées par le conférencier, ce nom a été récupéré lors de la phase1.

Etape 2. Après l'ouverture de la connexion, on ouvre un "canal" d'entrée qui va nous permettre de lire le fichier à télécharger. Ceci se fait en exécutant la méthode : url_image.**openConnection()**. On lit alors les données dans un *buffer* et on écrit dans un fichier dans le host local jusqu'à la fin du fichier.

Etape 3. Après on analyse le fichier pour en extraire les paquets *Whiteboard* c-à-d ceux qui ont le *Channel Number* à 2. Cette étape d'extraction nous permet de savoir les noms des slides que le conférencier utilisera dans sa conférence puisqu'on suppose que les participants auront une copie des slides avant le début de la conférence. Comme ce fichier contient toutes les actions du conférencier on a donc besoin d'un champ qui désigne les actions spécifiques au conférencier, c'est le champ **action** (6.5.1) qui joue ce rôle et pendant cette étape ce qui nous intéresse ce sont les paquets dont le champ action sont mis à 1025. Une fois le nom extrait, on ouvre une autre connexion de la même façon que dans l'Etape 2 et on lit le fichier image (de format GIF) et on l'enregistre dans le host local représentant le participant dans la conférence (c'est à dire l'auditeur).

Phase 3 : Cette phase représente le début de la conférence proprement dite.

Cette phase comporte 4 étapes :

Etape 1. On crée un thread d'exécution. Ce thread aura comme contexte l'*Applet* elle-même. Ceci se fait en invoquant la méthode suivante :
new Thread (**this**, " "). Cette méthode renvoie une instance de la classe **Thread**. **this** représente l'*Applet* elle-même.

Etape 2. Une fois le thread est crée, on lui alloue les ressources nécessaires pour son exécution et on invoque la méthode **run()** qui va mettre le thread dans l'état Exécutable.

Etape 3. On assigne la priorité **MIN_PRIORITY** à ce thread. Ceci se fait en invoquant la méthode **setPriortity**(Thread.MIN_PRIORITY).

Phase 4 : On décode le fichier en entrée pour en extraire les paquets *Whiteboard* dont le *Channel Number* est 2, le timestamp associé à l'action à exécuter, le type d'action ainsi que les coordonnées où il faut mettre le dessin sur l'écran. On ne s'occupe pas des paquets dont le *Channel Number* est à 1 puisque dans cet *Applet* on ne reproduit que l'action de l'utilisateur et non le discours du conférencier.

Phase 5 : On suspend le thread d'exécution pendant la période qui sépare le TIMESTAMP du dernier paquet décodé et celui du paquet actuellement décodé. Durant cette période c'est l'action du dernier qui est exécutée sur l'écran du client. De cette façon on lance l'action "exactement" au moment correspondant à l'action du conférencier.

Phase 6 : Après la terminaison de la période de suspension le thread d'exécution est réactivé et on revient à la Phase 4.

Les phases 4, 5 et 6 s'exécutent jusqu'à la fin du fichier c-à-d jusqu'à la fin du cours préenregistré.

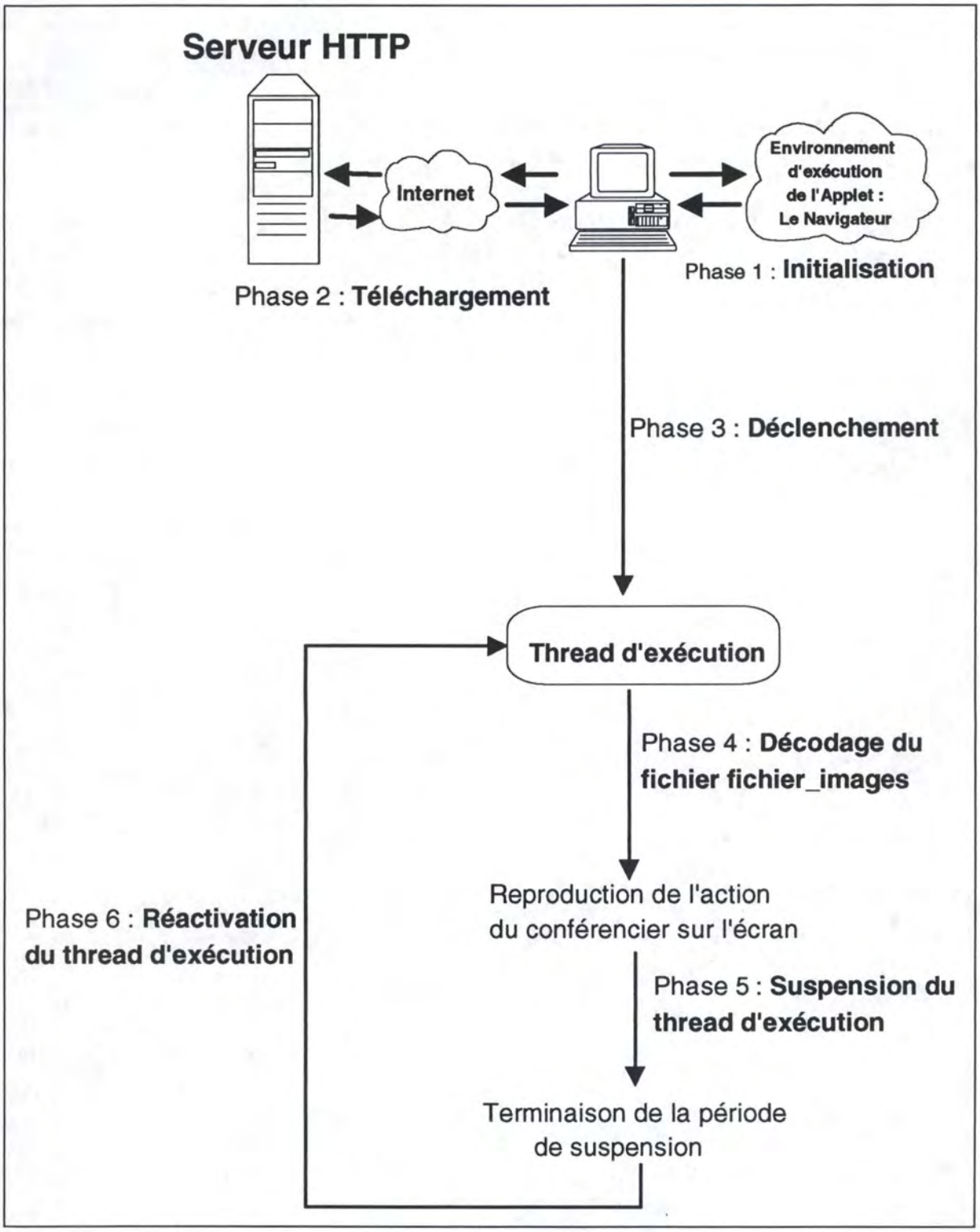


Figure 7.4 : L'*Applet* Reproduction_Action

7.4 L'*Applet* Reproduction_Action_Discours

La figure 7.5 montre les différentes phases par lesquelles passe l'*Applet* : **Reproduction_Action_Discours** :

Phase 1 : Dans cette première phase l'*Applet* initie une communication avec le navigateur qui représente son environnement d'exécution. Cette communication se traduit par la récupération de deux paramètres fournit par le navigateur et dont les valeurs représentent les noms des deux fichiers en entrées à savoir le fichier des actions et celui du son.

Cette communication se fait en mettant dans la page HTML, les étiquettes suivantes :

< **param name** = fichier_images **value** = "/bocage.données" >
< **param name** = fichier_son **value** = "/bocage.français" >

Ici la page HTML met à la disposition de l'*Applet* qui s'exécute dans son contexte deux paramètres à savoir fichier_images et fichier_son et dont les valeurs sont respectivement bocage.données et bocage.français.

Le fichier_images représente les actions de l'utilisateur et le fichier_son représente le discours du conférencier

L'*Applet* récupère dans le code les deux paramètres en exécutant les deux méthodes suivantes :

getParameter("fichier_images") et **getParameter**("fichier_son")

Ces deux méthodes renvoient à l'intérieur de l'*Applet* les deux valeurs : /bocage.données et /bocage.français.

Phase 2 : La phase 2 a pour but de télécharger du serveur le fichier : /bocage.données et d'en extraire les noms des slides qui vont être utilisés par le conférencier.
Pour cela on manipule une classe de haut niveau, à savoir la classe **URL**. URL est l'acronyme de Uniform Ressource Locator qui désigne une référence (ou une adresse d'une ressource sur Internet).

Cette phase comporte 3 étapes :

Etape 1. Ouverture d'une connexion vers le serveur en exécutant la méthode suivante :
new ("**http**", "www.info.fundp.ac.be","/bocage.données"), cette méthode renvoie une instance url_image de la classe **URL** qui est une classe fournit par JAVA pour ouvrir des connexions de haut niveau.

http représente le protocole HTTP qui dans le monde Internet a pour but de gérer l'échange d'une requête et d'une réponse entre un client et un serveur. "www.info.fundp.ac.be" représente le host auquel on se connecte qui fournit la page HTML et donc l'*Applet*.

/bocage.données représente le nom du fichier des actions exécutées par le conférencier, ce nom a été récupéré lors de la phase1.

Etape 2. Après l'ouverture de la connexion, on ouvre un "canal" d'entrée qui va nous permettre de lire le fichier à télécharger. Ceci se fait en exécutant la méthode : url_image.**openConnection()**. On lit alors les données dans un *buffer* et on écrit dans un fichier dans le host local jusqu'à la fin du fichier.

Etape 3. Après on analyse le fichier pour en extraire les paquets *Whiteboard* c-à-d ceux qui ont le champ *Channel Number* à 2. Cette étape d'extraction nous permet de savoir les noms des slides que le conférencier utilisera dans sa conférence puisqu'on suppose que les participants auront une copie des slides avant le début de la conférence. Comme ce fichier contient toutes les actions du conférencier on a donc besoin d'un champ qui désigne les actions spécifiques du conférencier, c'est le champ **action** qui joue ce rôle et pendant cette étape ce qui nous intéresse ce sont les paquets dont le champ action sont mis à 1025. Une fois le nom extrait, on ouvre une autre connexion de la même façon que dans l'Etape 2 et on lit le fichier image (de format GIF) et on l'enregistre dans le host local représentant le participant dans la conférence.

Phase 3 : Cette phase représente le début de la conférence proprement dit.

Cette phase comporte 4 étapes :

Etape 1. On crée le thread principal. Ce thread aura comme contexte l'*Applet* elle-même.
Ceci se fait en invoquant la méthode suivante : **new** Thread (**this**, " "). Cette méthode renvoie une instance de la classe **Thread**, le paramètre **this** représente l'*Applet*.

Etape 2. Une fois le thread est crée, on lui alloue les ressources nécessaires pour son exécution et on invoque la méthode **run()** qui va mettre le thread principal dans l'état Exécutable.

Etape 3. On assigne la priorité **MIN_PRIORITY** à ce thread. Ceci se fait en invoquant la méthode **setPriortity**(Thread.MIN_PRIORITY).

Phase 4 : Le thread principal crée deux autres threads, thread son et thread image, leur alloue les ressources systèmes et assignent à ces deux threads la priorité **MIN_PRIORITY**.

Ce sont ces deux threads qui décodent les deux fichiers en extrayant les paquets *Whiteboard* dont le champ *Channel Number* est mis à 2 pour le thread image et les paquets son dont le champ *Channel Number* est mis à 1 pour le thread son. Les deux threads renvoient au thread principal le TIMESTAMP de l'action (Timg) à exécuter pour le thread image et le TIMESTAMP de la donnée son (Tson) qu'il faut lancer sur la sortie audio. Par ailleurs le thread image renvoie les coordonnées où il faut mettre le dessin sur l'écran.

Phase 5 : Le thread principal compare les deux ***TIMESTAMPS*** correspondant aux deux threads et exécute la boucle d'exécution suivante :

si (time_stamp_image < time_stamp_son) **alors** le thread principal lance le thread image qui reproduit l'action du paquet décodé puisque le thread image renvoie aussi les coordonnées où il faut reproduire l'action sur l'écran. Le thread principal met en attente le thread son pendant la période séparant le timestamp du dernier paquet décodé et celui du paquet actuellement décodé.
si (time_stamp_son < time_stamp_image) **alors** le thread principal lance le thread son qui lance la donnée son du paquet décodé sur la sortie audio. Le thread principal met en attente le thread image pendant la période séparant le timestamp du dernier paquet décodé et celui du paquet actuellement décodé.
si (time_stamp_image = time_stamp_son) **alors** le thread principal lance les deux thread image et son.

Phase 6 : Après la terminaison du temps de suspension de l'un des deux threads, le thread principal reprend la main et exécute la phase 5 pour déterminer si c'est une action qu'il faut exécuter et c'est donc le thread image qui est activé ou si c'est le son qu'il faut lancer et c'est donc le thread son qu'il faut activer.

Les **phases 5 et 6** s'exécutent jusqu'à la fin des deux fichiers image et son c'est à dire la fin du cours préenregistré.

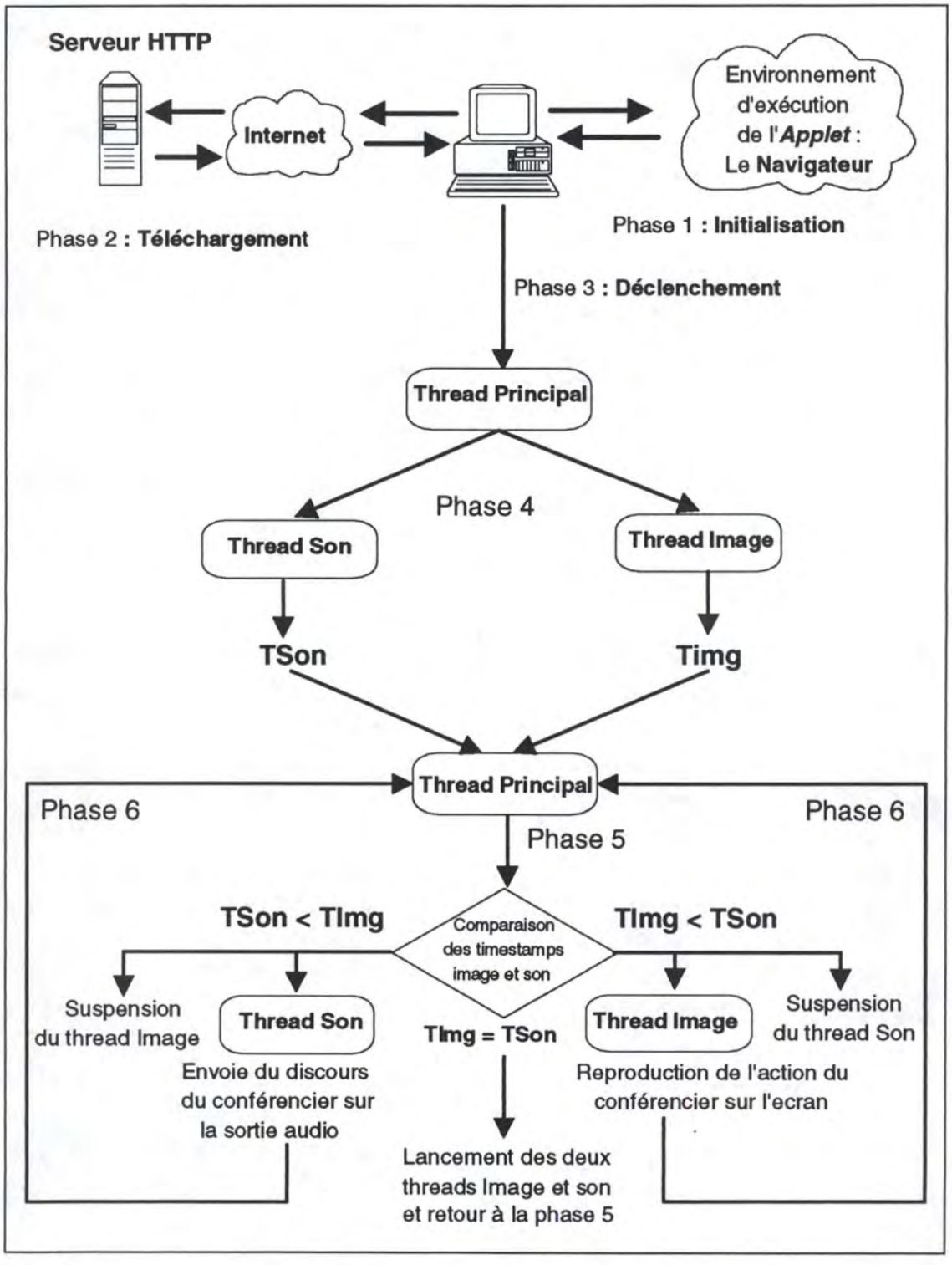


Figure 7.5 : L'Applet Reproduction_Action_Discours

CONCLUSION

Le mémoire avait deux objectifs :

1. Porter le mode ***slave*** **passif** de l'application *Whiteboard* sur le *Web*.
2. Présenter les architectures client/serveur, leurs évolutions et les tentatives pour faire du client/serveur sur le *Web*.

Pour le **premier objectif**, nous avons réalisé deux ***Applets*** JAVA. Le premier *applet* permet la reproduction des actions du conférencier sur l'écran de l'auditeur sans la reproduction du discours du conférencier. Le deuxième *applet* permet la reproduction aussi bien des actions du conférencier que de son discours. En utilisant un navigateur *Web* doté de l'environnement JAVA, chaque utilisateur peut télécharger une page HTML contenant le code compilé de l'*applet* et suivre un cours préenregistré.

Le **deuxième objectif** est lié au premier du fait que l'environnement de développement utilisé pour réaliser le premier objectif était JAVA. Pour réaliser le premier objectif on a utilisé les 3 fonctions principales de JAVA : **distribuer** des morceaux de code compilé sur le *Web*, **animer** les pages HTML et permettre une **interactivité** avec le client *Web*. Mais JAVA est aussi l'un des outils sur lequel va reposer certainement les stratégies de développement des applications client/serveur sur le *Web*. JAVA en tant que tel ne permet pas de faire du vrai client/serveur sur le Web vu les limites du protocole HTTP, mais combiné à d'autres technologies principalement la technologie des objets distribués, on peut développer de nouvelles applications client/serveur et porter les anciennes sur le *Web* tout en garantissant l'hétérogénéité qui caractérise le *Web*.

Perspective

Le mémoire peut être poursuivi dans trois directions :

La **première direction** concerne l'*applet* **Reproduction_Action_Discours**. On peut améliorer la synchronisation entre l'image et le son en exploitant deux voies : implémenter un module d'accélération de chargement des images ou utiliser des API temps-réel pour la Machine Virtuelle JAVA attendues dans les prochaines versions de l'environnement JAVA. Dans ce cadre on peut faire une étude des différences de performance entre différentes plate-formes supportant l'environnement JAVA et notamment : plate-forme Unix, Macintosh et Windows95/NT. Il est intéressant de voir comment ces différentes plate-formes gèrent le mécanisme des *threads* et la problématique du temps réel.

La **deuxième direction** est de donner la possibilité à l'utilisateur d'intervenir et donc d'avoir la possibilité de passer du mode *slave* vers le mode *master* et de ce fait lui permettre d'accomplir du dessin et de donner son avis. Si on veut toujours utiliser les opportunités offertes par le *Web*, on peut faire ceci en implémentant un autre *applet* représentant le mode *master* qui représente le *Whiteboard* en mode *master*. On aura alors deux *applets* : l'*applet* représentant le mode *slave* passif dans lequel on gère les événements externes notamment ceux émanant de l'utilisateur et l'*applet* représentant le mode *master* du *Whiteboard*. La communication entre ces deux *applets* peut se faire de trois façons : mettre les deux *applets* dans la même page HTML. L'utilisateur peut alors charger une seule page HTML qui contiendra les deux *applets* (l'*applet* mode *master* et l'*applet* mode *slave*). Dans ce cas la communication peut se faire en utilisant des API fournit par l'environnement JAVA telles **AppletContext** et **AppletStub** qui permettent de passer le nom de l'*applet* appelé (l'*applet master*) à l'*applet* appelant (mode *slave*). Ceci permettra à l'*applet slave* d'invoquer des méthodes de l'*applet master* comme par exemple demander au mode *master* d'avoir les outils de dessin. La deuxième façon de faire est d'utiliser un *applet* intermédiaire qui jouera le rôle de variable partagée et qui aura pour rôle de gérer la communication entre les deux *applets*. Une troisième façon de faire est d'utiliser le ***Remote Method Invocation*** (RMI) pour gérer la communication entre deux objets JAVA appartenant à deux machines virtuelles différentes. Dans ce cas les deux *applets* ne sont pas nécessairement chargés à partir de la même page HTML.

La **troisième direction** c'est d'utiliser les autres technologies de développement sur le *Web*, notamment la technologie *ActiveX* (OLE *controls*) et l'Internet *Explorer*, les deux produits qui représentent l'infrastructure Internet de Microsoft, ou encore la technologie CORBA d'OMG et son bus logiciel ORB. Il est aussi intéressant de voir le niveau de performance entre JAVA, OLE/DCOM et CORBA.

Table des figures

GLOSSAIRE

Applet : Un programme écrit en JAVA qui s'exécute dans le contexte d'une page HTML.

Bytes-code : Code généré lors de la compilation du code JAVA. Ce code ne contient aucun détail d'implémentation et donc neutre par rapport à une architecture particulière. Ce code est exécuté par l'interpréteur JAVA.

CGI : **C**ommon **G**ateway **I**nterface. Le CGI permet à un client *Web* de lancer et de passer de l'information à un programme. Le client *Web* rempli un formulaire et c'est le serveur HTTP qui lance le script CGI avec les informations remplies. En retour le script CGI crée dynamiquement une page HTML qui est alors transmise au client *Web*.

HTML : **H**yper**T**ext **M**arkup **L**anguage. Les pages HTML représentent les documents communément manipulés sur le *Web*. Ces pages peuvent inclure des images, du son, du vidéo, des formulaires à remplir par l'utilisateur ou des liens vers d'autres pages HTML.

HTTP : **H**yper**T**ext **T**ransfer **P**rotocol. Il permet l'échange d'une **requête** et d'une **réponse** entre un client HTTP et un serveur HTTP. Il est basé sur le protocole TCP/IP.

IDL : **I**nterface **D**efinition **L**anguage. C'est un langage purement déclaratif qui définit l'interface d'un objet en terme d'attributs, de méthodes et de paramètres sans aucun détail d'implémentation. Il permet d'isoler l'interface d'un objet de son implémentation.

JAVA : JAVA est un langage de programmation interprété permettant d'écrire des applications JAVA appelées *Applets* capables de s'exécuter dans le contexte d'une page HTML.

Mode *master* du Whiteboard : Permet à un utilisateur de disposer des outils de dessin, de contrôler le déroulement d'une conférence et de diffuser les informations de dessin sur le réseau.

Mode *slave* du Whiteboard : Permet à un utilisateur de recevoir l'information transmise par le Whiteboard en mode master et de l'afficher localement.

NCJ : **N**avigateur **C**onforme à **J**AVA. C'est tout navigateur disposant de l'interpréteur JAVA et de son environnement d'exécution. Les versions actuelles de Netscape représentent un exemple de NCJ.

PRISM : **P**late-forme **R**épartie pour l'**I**ntégration de **S**ervices **M**ultimédia. Il permet le développement d'application multimédia dans un environnement hétérogène.

RMI : **R**emote **M**ethod **I**nvocation. RMI permet à un objet JAVA d'invoquer la méthode d'un autre objet JAVA distant. Chaque objet est décrit par une ou plusieurs interfaces. RMI représente le modèle objets distribués utilisant les interfaces JAVA.

RPC : **R**emote **P**rocedure **C**all. Le RPC permet à un appel de procédure au niveau d'un site client d'être converti en un appel de procédure au niveau d'un site serveur distant et ceci de façon transparente sans prendre en compte les détails concernant l'architecture du réseau.

Whiteboard : Un outil de haut niveau du bloc multimédia de la plate-forme PRISM. Sa fonction est de fournir un espace de travail commun pour un ensemble d'utilisateurs. Il peut être activé selon deux modes : le mode *master* et le mode *slave*.

Bibliographie

[Hal96] Tom R. HALFHILL, Salvatore SALAMONE, "Components Everywhere" BYTE, January 1996, p : 97-102.

[Hub94] Olivier Joseph HUBER, The *WHITEBOARD* : A High Level Multimedia Service for the PRISM platform, École Nationale Supérieure de Télécommunication de Bretagne Rennes, 1994.

[Orf96] Robert ORFALI, Dan HARKEY, Jeri EDWARDS, The Essential Distributed Objects Survival Guide, John Wiley & Sons, Inc., 1996.

[Ple96] Keith PLEAS, "OLE's Missing Links", BYTE, Avril 1996, p : 99-102.

[Sun95] Documentation des laboratoires Sun disponible sous l'url, http://java.sun.com

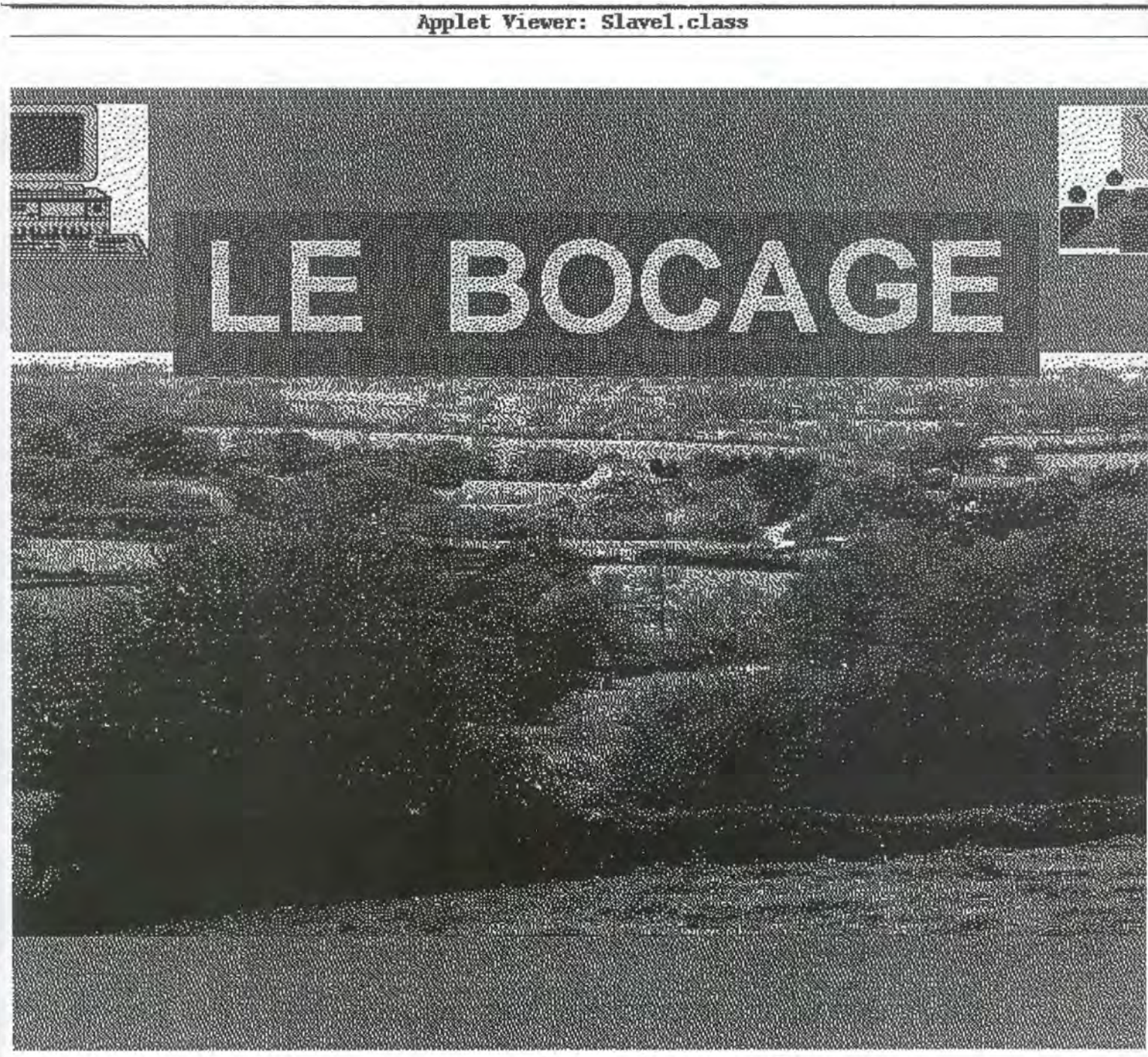
[Uma93] Amjad UMAR, Distributed Computing and Client-Server Systems, Prentice-Hall, Inc., 1993.

[Van95] Michel VAN ASTEN, Contribution au projet PRISM : Amélioration du *Whiteboard*, un outil multimédia de haut niveau, Institut d'Informatique, FUNDP, NAMUR, 1995.

[Van96] Arthur VAN HOFF, Sami SHAIO, Orca STARBUCK, HOOKED on JAVA, Addison-Wesley Publishing Company, 1996.

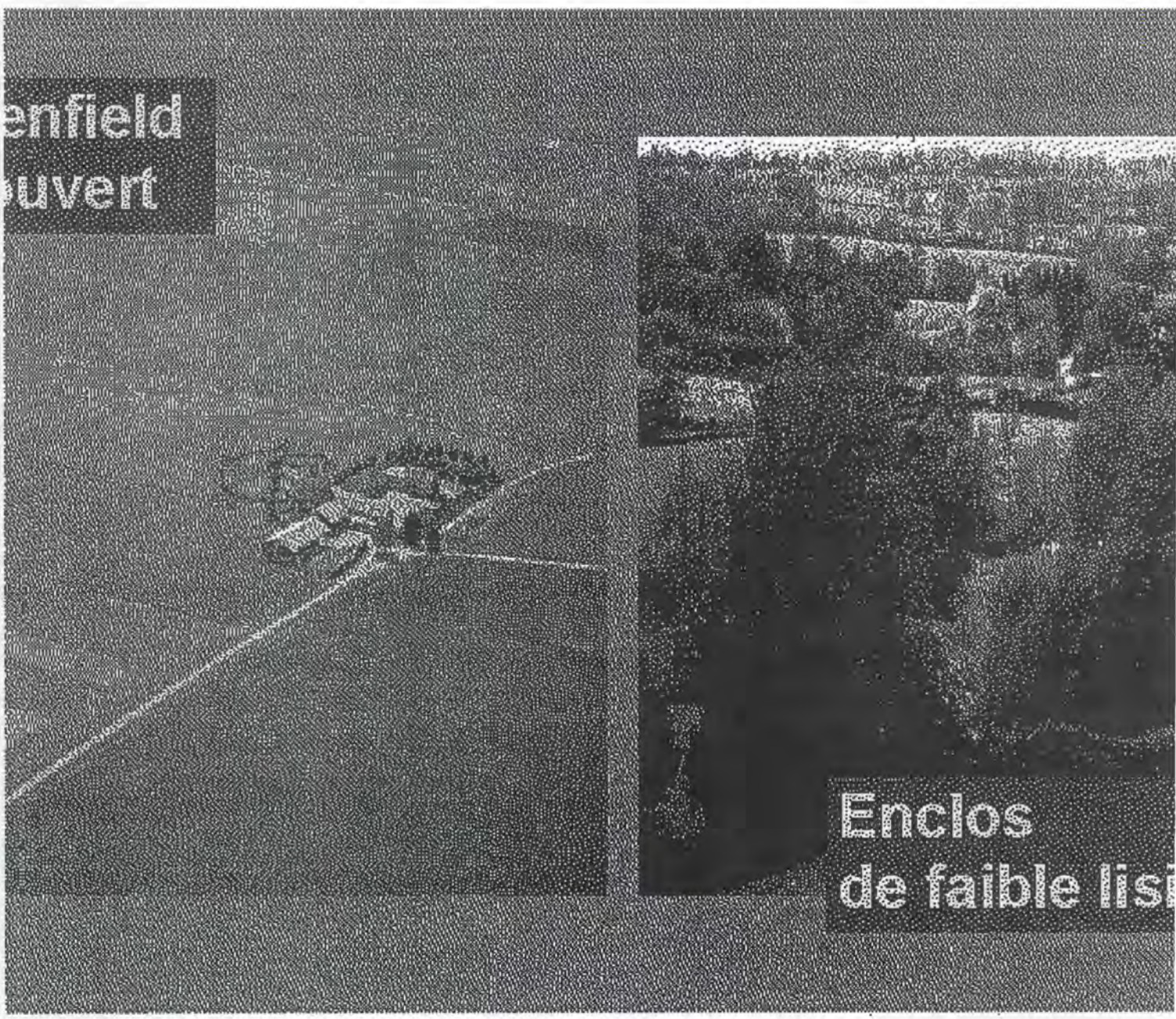
ANNEXE

Dans cet annexe on présente des écrans de l'*applet* **Reproduction_Action**.
Les écrans de l'*applet* **Reproduction_ Action_Discours** sont les mêmes puisqu'il ne diffère du premier que par l'introduction du son.
L'écran ci-dessous représente le début de la conférence (ou du cours) enregistré. La première action représente l'affichage du premier slide (image de format **gif**). Au niveau de l'interface, un indicateur montrant la progression du cours, est introduit.

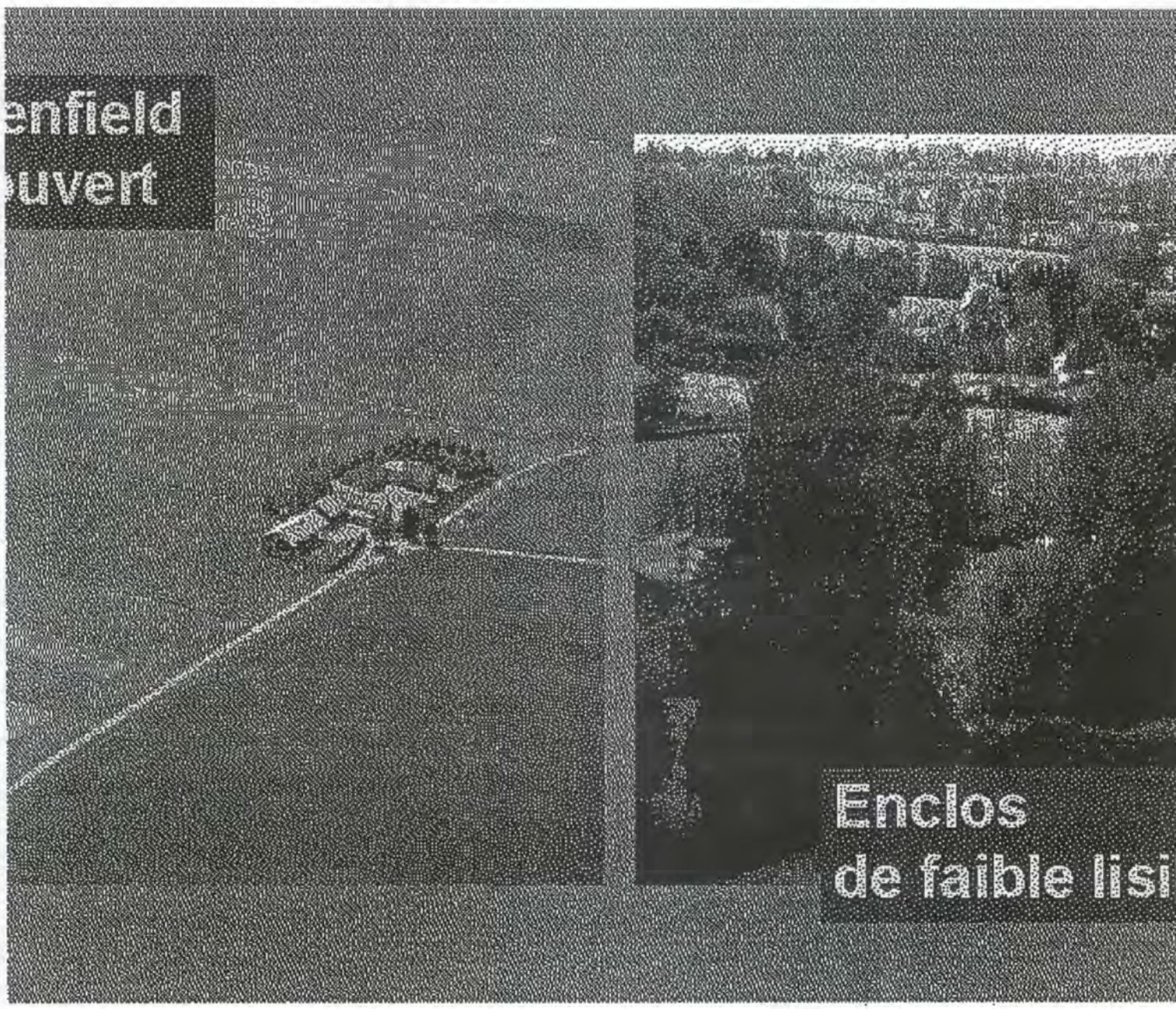


L'écran ci-dessous représente l'action du conférencier indiquant un point précis du slide. Ceci se traduit au niveau de l'*applet* par la superposition de l'image représentant le slide et celle représentant la main du conférencier. Cette dernière est placée à l'endroit correspondant aux coordonnées extraites du paquet. L'action du conférencier est reproduite au moment correspondant au *timestamp* extrait du paquet.

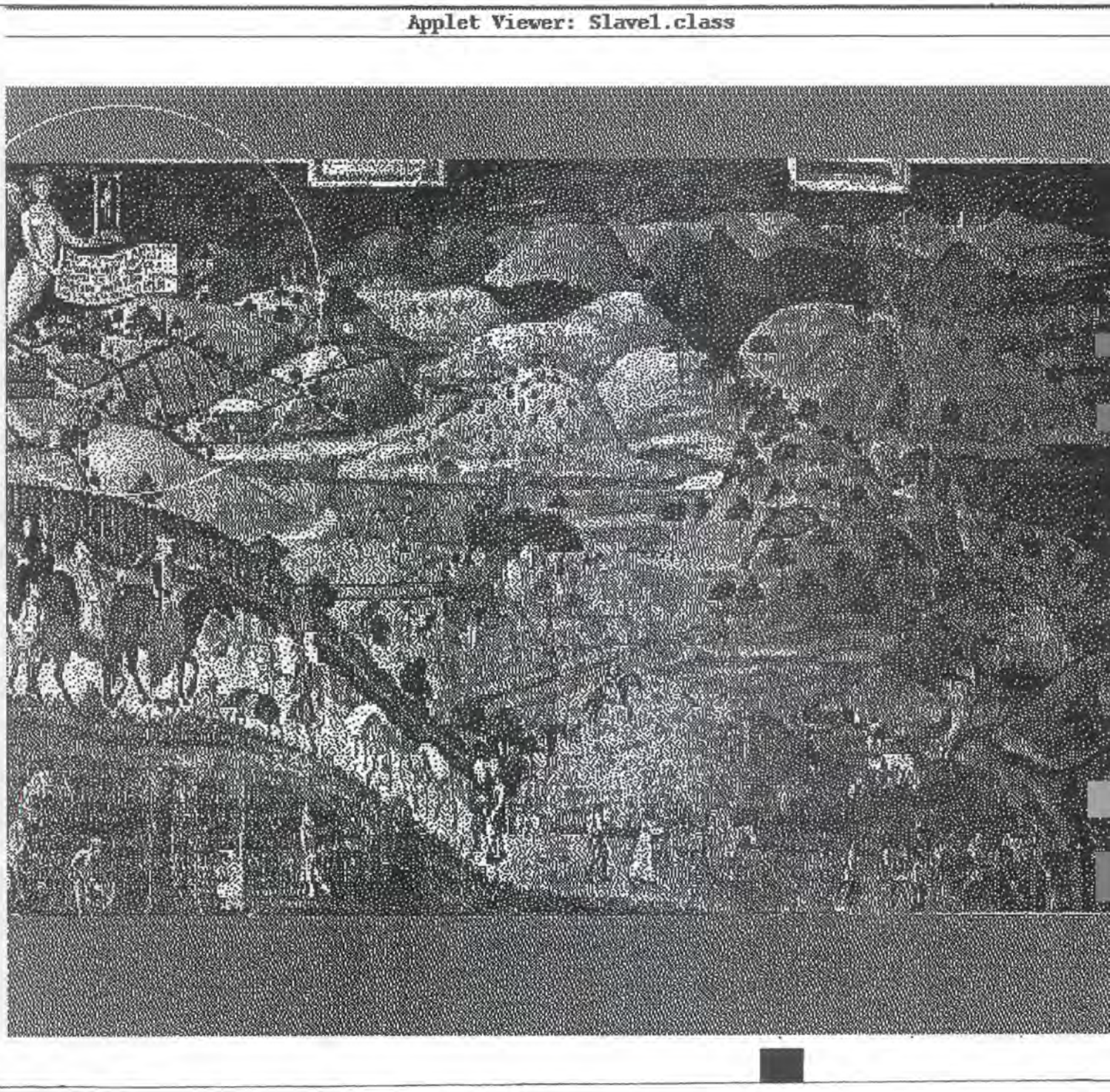
Applet Viewer: Slave1.class



L'écran ci-dessous représente la même image précédante mais maintenant le conférencier indique un autre endroit du slide à un autre *timestamp*.



L'écran ci-dessous représente une autre action du conférencier indiquant une région du slide en traçant un cercle. Au niveau de l'interface, l'indicateur est avancé suivant le nombre de slides restant.



Le dernier écran représente la terminaison du cours. La position de l'indicateur montre aussi la fin du cours.

Applet Viewer: Slave1.class